

TADS 3 TOUR GUIDE

FOR TADS 3.0.12

Eric Eve

CONTENTS

1. Introduction	8
1.1. General Introduction	8
1.2. The Sample Game	9
1.3. Templates	9
1.4. Startup Code : gameMain	15
2. Rooms and Connectors	18
2.1. Introduction	18
2.2. OutdoorRoom	19
2.3. FakeConnector	20
2.4. DeadEndConnector	21
2.5. RoomConnector	22
2.6. asExit	23
2.7. Enterable	23
2.8. Room	24
2.9. StairwayDown	25
2.10. StairwayUp	25
2.11. TravelWithMessage	26
2.12. SecretDoor	27
2.13. ThroughPassage	28
2.14. DarkRoom	29
2.15. TravelMessage	29
2.16. RoomAutoConnector	30
2.17. Door	30
2.18. BasicDoor	31
2.19. NoTravelMessage	31
2.20. ExitOnlyPassage	32
2.21. AutoClosingDoor	32
2.22. OneWayRoomConnector	34
2.23. PathPassage	34
2.24. Shipboard	36
2.25. FloorlessRoom	38
2.26. Floorless	39
2.27. HiddenDoor	39
2.28. EntryPortal	41
2.29. ExitPortal	41
2.30. TravelBarrier	42
2.31. AskConnector	42
2.32. TravelConnector	43
2.33. Room Methods and Properties	47
2.33.1. roomXxxxAction	47
2.33.2. roomParts	48
2.33.3. cannotGoThatWay	50
2.33.4. cannotGoThatWayInDark	51
2.33.5. roomDarkTravel	52
2.33.6. enteringRoom	52
2.33.7. inRoomName	53
3. NonPortables	54
3.1. NonPortable Introduction	54
3.2. Fixture	55
3.3. CustomFixture	55
3.4. Decoration	55
3.5. Distant	57
3.6. Unthing	58
3.7. Immovable	59
3.8. CustomImmovable	60
3.9. Heavy	60
3.10. Component	60
4. Things	62
4.1. Thing - Introduction	62
4.2. Thing - The Basics	62
4.3. vocabWords	63
4.4. initDesc & initSpecialDesc	65
4.5. globalParamName	66

TADS 3 Tour Guide

4.6.	specialDesc	66
4.7.	described	69
4.8.	bulk and weight.....	70
4.9.	setSuperclassList.....	70
4.10.	Readable	71
4.11.	Food	72
4.12.	disambigName	72
4.13.	Wearable	73
5.	Containers.....	74
5.1.	Containers - Introduction	74
5.2.	BulkLimiter	74
5.3.	Surface.....	75
5.4.	BasicContainer	75
5.5.	Container	76
5.6.	OpenableContainer.....	77
5.7.	notifyInsert & notifyRemove.....	77
5.8.	LockableContainer	79
5.9.	RestrictedContainer	81
5.10.	Dispenser	84
5.11.	StretchyContainer	85
5.12.	SpaceOverlay.....	85
5.13.	Underside	86
5.14.	RearContainer	87
5.15.	RearSurface	89
5.16.	ComplexContainer	90
5.17.	ContainerDoor.....	92
5.18.	SingleContainer.....	93
5.19.	BagOfHolding.....	94
6.	Locks & Keys	95
6.1.	Locks & Keys - Introduction	95
6.2.	Lockable.....	95
6.3.	IndirectLockable.....	96
6.4.	KeyedContainer	97
6.5.	LockableWithKey	99
6.6.	Keyring.....	100
6.7.	Openable	100
6.8.	BasicOpenable	101
7.	Light and Fire	102
7.1.	Light and Fire - Introduction.....	102
7.2.	brightness	102
7.3.	LightSource.....	102
7.4.	Flashlight.....	103
7.5.	Candle & FireSource	103
7.6.	OilLamp	105
7.7.	Matchstick & Matchbook.....	107
7.8.	Dynamite.....	108
8.	Hiding & Finding.....	110
8.1.	Hiding & Finding - Introduction	110
8.2.	Hiding with Words.....	110
8.3.	Finding by moving.....	111
8.4.	sightPresence & isListed	112
8.5.	Hidden.....	113
8.6.	PresentLater	114
9.	Gadgets & Controls.....	116
9.1.	Gadgets - Introduction	116
9.2.	Button.....	116
9.3.	LabeledDial	117
9.4.	SpringLever	117
9.5.	Settable.....	118
9.6.	NumberedDial.....	119
9.7.	Dynamic Locations	119
9.8.	Lever	123
9.9.	Dial.....	127
9.10.	OnOffControl	129
9.11.	Switch.....	130

TADS 3 Tour Guide

9.12.	Lever (2)	131
9.13.	A Card Lock	133
10.	Fuses & Daemons	135
10.1.	Fuse	135
10.2.	Daemon	137
10.3.	SenseFuse	140
10.4.	SenseDaemon	142
10.5.	PromptDaemon	142
10.6.	OneTimePromptDaemon	142
11.	ModuleExecObjects	144
11.1.	ModuleExecObject	144
11.2.	InitObject	144
11.3.	PreinitObject	145
11.4.	PreSaveObject	146
11.5.	PostRestoreObject	146
11.6.	PostUndoObject	147
11.7.	PreRestartObject	147
12.	Pushing Things Around	149
12.1.	TravelPushable	149
12.2.	PushTravelBarrier	151
13.	Intangibles & Senses	153
13.1.	Intangibles - Overview	153
13.2.	Intangible	153
13.3.	DistanceConnector	153
13.4.	Occluder	155
13.5.	Vaporous	157
13.6.	SimpleOdor	158
13.7.	SimpleNoise	160
13.8.	Odor	160
13.9.	Noise	162
13.10.	SenseConnector	164
13.11.	SensoryEvent	165
14.	Attachables	170
14.1.	Attachables - Overview	170
14.2.	Attachable	170
14.3.	NearbyAttachable	174
14.4.	PlugAttachable	176
14.5.	PermanentAttachment	177
15.	NestedRoom	179
15.1.	NestedRoom Overview	179
15.2.	NestedRoom	179
15.3.	BasicChair	180
15.4.	Platform	180
15.5.	NominalPlatform	181
15.6.	Bed	183
15.7.	Chair	183
15.8.	HighNestedRoom	184
15.9.	OutOfReach	185
15.10.	Booth	187
15.11.	Vehicle	189
15.12.	VehicleBarrier	191
16.	MultiLocs	195
16.1.	MultiLoc	195
16.2.	MultiInstance	196
16.3.	MultiFaceted	197
17.	Collections	199
17.1.	CollectiveGroup (static)	199
17.2.	CollectiveGroup (mobile)	202
18.	Scripts	205
18.1.	Script	205
18.2.	EventList	205
18.3.	StopEventList	207
18.4.	CyclicEventList	207
18.5.	RandomEventList	209
18.6.	ShuffledEventList	210

TADS 3 Tour Guide

18.7.	ExternalEventList	210
18.8.	SyncEventList	210
18.9.	RandomFiringScript	211
19.	Actors & NPCs	212
19.1.	Overview - Actors & NPCs	212
19.2.	Basic Actors	212
19.3.	Basic Actor Customization	214
19.4.	Actor Knowledge	215
19.5.	Moving Actors Around	216
19.6.	Actor States	216
19.6.1.	Overview - Actor States	216
19.6.2.	HermitActorState	218
19.6.3.	AccompanyingState	218
19.6.4.	AccompanyingInTravelState	219
19.6.5.	GuidedTourState	220
19.6.6.	GuidedInTravelState	222
19.6.7.	InConversationState	222
19.6.8.	ConversationReadyState	223
19.6.9.	Greeting Protocols	225
19.7.	Topic Entries	226
19.7.1.	TopicEntry	226
19.7.2.	GiveTopic	229
19.7.3.	ShowTopic	230
19.7.4.	GiveShowTopic	232
19.7.5.	AltTopic	235
19.7.6.	InitiateTopic	236
19.7.7.	AskTopic	240
19.7.8.	TellTopic	243
19.7.9.	AskTellTopic	244
19.7.10.	AskForTopic	245
19.7.11.	AskAboutForTopic	248
19.7.12.	AskTellShowTopic	249
19.7.13.	AskTellGiveShowTopic	249
19.7.14.	Yes,No & SpecialTopics	250
19.7.15.	HelloTopic	250
19.7.16.	ImpHelloTopic	251
19.7.17.	ByeTopic	251
19.7.18.	ImpByeTopic	252
19.7.19.	LeaveByeTopic	252
19.7.20.	BoredByeTopic	252
19.7.21.	ActorByeTopic	252
19.7.22.	HelloGoodbyeTopic	253
19.7.23.	MiscTopic	253
19.7.24.	TopicGroup	256
19.7.25.	DefaultTopics	257
19.7.26.	SuggestedTopics	267
19.8.	Conversation Nodes	274
19.8.1.	Conversation Nodes - Overview	274
19.8.2.	ConvNode	275
19.8.3.	YesTopic	277
19.8.4.	NoTopic	277
19.8.5.	SpecialTopic	277
19.8.6.	initiateConversation	279
19.9.	AgendaItems	285
19.9.1.	AgendaItem	285
19.9.2.	ConvAgendaItem	287
19.9.3.	DelayedAgendaItem	288
19.9.4.	More AgendaItem Examples	289
19.10.	Commanding NPCs	292
19.10.1.	Overview - Commanding NPCs	292
19.10.2.	CommandTopic	293
19.10.3.	DefaultCommandTopic	294
19.10.4.	Overriding obeyCommand	295
19.10.5.	TCommandTopic	296
19.10.6.	A Modified DefaultCommandTopic	299

TADS 3 Tour Guide

20.	Consultables	301
20.1.	Consultable	301
20.2.	ConsultTopic	303
20.3.	DefaultConsultTopic	304
21.	Scoring	306
21.1.	Scoring - Overview	306
21.2.	addToScore	306
21.3.	Achievement	307
21.4.	SimpleAchievement	310
21.5.	awardPoints	310
21.6.	awardPointsOnce	311
21.7.	scoreRankTable	311
21.8.	maxScore	312
22.	Hints	314
22.1.	Hints - Overview	314
22.2.	TopHintMenu	314
22.3.	HintMenu	314
22.4.	Goal	315
22.5.	Hint	318
23.	Further Information	320
23.1.	Concluding Remarks	320
23.2.	Language Information	320
23.3.	Defining Verbs	320
23.4.	Message Substitution Parameters	320
23.5.	Past Tense	321
24.	Templates	322
24.1.	Achievement Template	322
24.2.	Actor Template	322
24.3.	AltTopic Template	322
24.4.	ConvNode Template	322
24.5.	DeadEndConnectorTemplate	322
24.6.	DefaultTopic Template	322
24.7.	Enterable Template	323
24.8.	Exitable Template	323
24.9.	EventList Template	323
24.10.	Footnote Template	323
24.11.	Goal Template	323
24.12.	Hint Template	323
24.13.	MenuItem Template	323
24.14.	MenuLongTopicItem Template	324
24.15.	MenuTopicItem Template	324
24.16.	MiscTopic Template	324
24.17.	MultiLoc Template	324
24.18.	NoTravelMessage Template	324
24.19.	OneWayRoomConnector Template	324
24.20.	Passage Template	324
24.21.	Room Template	325
24.22.	ShuffledEventList Template	325
24.23.	SpecialTopic Template	325
24.24.	StyleTag Template	325
24.25.	SyncEventList Template	325
24.26.	Thing Template	325
24.27.	ThingState Template	326
24.28.	TopicEntry Template	326
24.29.	TopicGroup Template	326
24.30.	TravelMessage Template	326
24.31.	Unthing Template	327
24.32.	VocabObject Template	327
25.	Changes	328
25.1.	Changes for v3.0.12	328
25.2.	Changes for v3.0.10	328
25.3.	Changes for v3.0.9	328
25.4.	Changes for v3.0.8	329
25.5.	Changes for July-Sept 2004	329
25.6.	Changes for v3.0.7	329

TADS 3 Tour Guide

25.7.	Changes for v.3.0.6q.....	330
25.8.	Changes for May 2004.....	330
25.9.	Changes for v3.0.6p.....	330
25.10.	March/April 2004	330
INDEX	331

1. Introduction

1.1. General Introduction

The adv3 library that comes with the TADS 3 Interactive Fiction authoring system is extensive and powerful. It can also seem rather overpowering to new users of TADS 3, because there is so much to learn, and one hardly knows where to start looking for what one needs.

Enter the *TADS 3 Tour Guide*. Its aim is to give a guided tour of some of the main features of the TADS 3 library. We shall not be exploring every nook and cranny (at this stage it would probably be more confusing than helpful to do so). Nor shall we be able to wander down every sidestreet and alley, though we may poke our noses into a few. What we shall aim to do is to walk round most of the main streets so that their basic layout and interconnections will hopefully start to become clear.

This Tour Guide is *not* intended as the first port of call in learning TADS 3. If you are a complete beginner I strongly recommend you start with my introductory *Getting Started in TADS 3 : A Beginner's Guide*, which you can download from <http://www.tads.org> (or which you may already have with your TADS 3 distribution). Although there will be some overlap with material covered there, the *Tour Guide* assumes basic familiarity with programming in the TADS 3 language and the definition of simple TADS 3 objects. For this Guide is *not* a TADS 3 manual, or a substitute for one. Neither is it an introduction to writing games in TADS 3, or an exhaustive description of every Class, property and method in the TADS 3 library. Finally, it is *not* a complete guide to the TADS language, many features of which are already well documented in the *System Manual* that come with the TADS 3 distribution, or which can be downloaded from <http://www.tads.org>.

What this *Tour Guide* is is a kind of Guided Tour to the TADS 3 library, that tries to take in as many as possible of the classes, properties and methods that are likely to be useful to most game authors. The assumption is that what you will find useful is not so much a load of abstract explanation, but rather a series of concrete examples. This *Tour Guide* therefore takes you through developing a sample game, introducing each Class in turn with one or two examples of its use. Later sections re-use classes and properties introduced before, and sometimes suggest further sophistications.

This *Guide* may thus be used either as a tutorial or as a reference (or both). As a tutorial it may be worked through from start to finish, developing the game step-by-step until all the main features of the library have been introduced and exemplified; you may like to use it as a follow-on tutorial from *Getting Started in TADS 3*. But it may also be used as a reference to the use of various library classes. For the latter purpose the Windows help file version of this *Guide* is likely to prove most useful; for the former you may prefer the PDF version. Note, however, that the *Tour Guide* will probably be more useful as a reference once you have worked through it as a tutorial, since in the very nature of its treatment of developing a game, its later sections presuppose objects and concepts mentioned in earlier sections, and many techniques have to be introduced in passing. Note also that a complete reference to the library is provided by the *TADS 3 Library Reference Manual*.

Of necessity there must be some compromise between the need to develop the sample game in a reasonably logical sequence and the desirability of presenting the various library classes in a reasonably logical sequence. For this reason we start by looking at Rooms and Connectors, since laying out some kind of map is necessary before anything much else can happen in a game. We then go through the other Classes representing concrete game objects, before going on to look at the creation of NPCs and the use of more abstract classes for conversations, scoring, hints and the like.

IMPORTANT NOTE - This Guide is intended for use with **TADS 3.0.12**. Although changes from the immediately preceding versions of TADS 3 (especially TADS 3.0.10 and later) are relatively minor prior, successive library updates have substantial changes; if you are using a significantly older version of TADS 3 a great deal in this guide **will not work**. Please therefore:

- Update to TADS 3.0.12 before attempting to work through this Guide. See <http://www.tads.org/t3dl.htm>.
- If, for some reason, you are unable to update to TADS 3.0.12, please be aware that any problems you encounter may be due to incompatibilities between versions of the library rather than bugs in the sample code.

Finally, I hope this Tour Guide will prove helpful, and even enjoyable to use. I always welcome feedback and suggestions, not least those that point out genuine errors, typos or bugs. I can be contacted by email on *eric dot eve*

Eric Eve
03-Sep-06

1.2. The Sample Game

The Sample Game we shall be developing together is whimsically called *The Quest of the Golden Banana*. There is nothing stunningly original in its design, and it will probably strike you as Dr Who meets the Lord of the Rings with a hint of old-fashioned text adventures like *Zork* thrown in for good measure. For our purposes, however, its contrived nature is an advantage, since it allows us to set up of sorts of implausible situations and puzzles that can put the TADS 3 library through its paces.

To give a brief summary of the game (which you might want to try playing before getting to work on this Tour Guide), you (i.e. the Player Character) start outside the entrance of a cave. The only way to go is in, and once you're in there's a rockfall that blocks the only obvious way out. To complete the game you need to fulfil two objectives: (1) locate the Golden Banana and cast it into Mount Gloom to prevent its falling into the hands of the dreaded Cabal, and (2) help the young woman you'll quickly encounter to get back out of the cave (i.e. both you and the woman must arrive back at the starting location). To fulfil the first objective you need to sail a ship round an underground lake to various destinations; to fulfil the second you need to get a TARDIS back in full working order: you'll also need the TARDIS to fulfil the first objective, traveling to locations as disparate as King Solomon's palace three thousand years in the past to an abandoned space station a thousand years in the future.

1.3. Templates

Since we shall be using templates extensively to define objects throughout this Guide, we had better start by explaining what they are and how they work.

If you have worked through *Getting Started in TADS 3* or some similar introductory guide, you'll already have encountered templates. Templates are built into the TADS 3 language (in the sense that the language provides the facility to define and use them) and into the adv3 library (in the sense that the library defines a number of standard templates). This Tour Guide accordingly assumes that the use of templates is the standard TADS 3 coding style, and is to be encouraged. But first, what exactly is a template?

Put simply, a template is a means of defining an object in a more succinct form in order to save typing effort and produce less verbose code. On the principle that a couple of examples are a good deal easier to follow than several paragraphs of abstract, theoretical discussion, we'll explain this by showing how templates work with the most common kind of objects you're likely to define in a TADS 3 game: Rooms and Things.

We'll start with a room. If we defined a Room without using a template, we should have to assign every property we wanted assigned explicitly. Such a definition might look like:

```
entranceCave : Room
    roomName = 'Entrance Cave'
    destName = 'the entrance cave'
    desc =
        "Compared with the narrow tunnel leading out to the north, this
        rough-walled cave seems positively spacious. A red sign fixed to
        one wall suggests that the narrow tunnel is the only way back out to
        the valley, while a blue sign next to it welcomes you to the cave.
        Directly under the signs a narrow ledge has been carved into the
        wall. There appear to be no other caves at this level, but a sturdy
        steel ladder leads down through a large round hole in the floor. "

    north = entranceTunnel
    out asExit(north)
;
```

Taking advantage of the Room template, the same definition could be coded as:

TADS 3 Tour Guide

```
entranceCave : Room 'Entrance Cave' 'the entrance cave'
    "Compared with the narrow tunnel leading out to the north, this
    rough-walled cave seems positively spacious. A red sign fixed to
    one wall suggests that the narrow tunnel is the only way back out to
    the valley, while a blue sign next to it welcomes you to the cave.
    Directly under the signs a narrow ledge has been carved into the
    wall. There appear to be no other caves at this level, but a sturdy
    steel ladder leads down through a large round hole in the floor. "

    north = entranceTunnel
    out asExit(north)
;
```

These two definitions are entirely equivalent; both assign exactly the same values to the same properties.

So how does this work? The Room template is defined in the library as follows:

```
Room template 'roomName' 'destName'? 'name'? "desc"?
```

This definition means that when defining an object of class Room (or one of its subclasses), if the class name is immediately followed by a single-quoted string, that string will be assigned to the **roomName** property; the next single-quoted string, if present, will be assigned to the **destName** property, the next to the **name** property, and a double-quoted string that comes at the end of this list will be assigned to the **desc** property. The question mark after an item in a template definition means that this element is optional and may be omitted.

Accordingly, the following definitions using the Room template are all legal:

```
entranceCave : Room 'Entrance Cave'
;
```

Which is equivalent to:

```
entranceCave : Room
    roomName = 'Entrance Cave'
;
```

Or

```
entranceCave : Room 'Entrance Cave'
    "Compared with the narrow tunnel..."
;
```

Which is equivalent to:

```
entranceCave : Room
    roomName = 'Entrance Cave'
    desc = "Compared with the narrow tunnel..."
;
```

Or

```
entranceCave : Room 'Entrance Cave' 'the entrance cave'
;
```

Which is equivalent to:

```
entranceCave : Room
    roomName = 'Entrance Cave'
    destName = 'the entrance cave'
;
```

Or

```
entranceCave : Room 'Entrance Cave' 'the entrance cave' 'entrance cave'
;
```

Which is equivalent to:

TADS 3 Tour Guide

```
entranceCave : Room
    roomName = 'Entrance Cave'
    destName = 'the entrance cave'
    name = 'entrance cave'
;
```

Note, however, that properties defined in the template must appear in the order shown, so that the following would *not* match the template:

```
entranceRoom "Compared with the narrow tunnel..." 'Entrance Cave';
entranceRoom 'Entrance Cave' "Compared with the narrow tunnel..." 'the entrance cave';
```

In practice, virtually all rooms will need to define a roomName and a description (and this is the point of the template, to allow the common properties of all rooms to be defined with the minimum of effort). So you will normally define rooms in one of two forms:

```
myRoom : Room 'My Room Name'
    "My room desc "
    /* other properties/methods */
;
```

or

```
myRoom : Room 'My Room Name' 'my room destName'
    "My room desc "
    /* other properties/methods */
;
```

Not only does this make defining rooms *briefier*, it also makes your code more readable, since the key properties (roomName, destName if defined, and desc) will always appear in the same relative location in the definition of a room, rather than at some possibly random location in a list of properties (for these key properties will seldom be the *only* properties you'll need to define on a room). Once you get used to the template, you can look at a room definition and see its roomName and description at once.

Note that a template defined for a class is also valid for all its subclasses. So the [Room](#) template we have just described can (and should) also be used for [OutdoorRoom](#), [DarkRoom](#) and [FloorlessRoom](#) (and, indeed, for any specialized subclasses of Room you may define in your own game).

Now let's look at the definition of the Thing template (which also applies to all the subclasses of Thing, i.e. virtually every game object that represents a physical object in the game world, unless there's a more specific template applying to the subclass).

The Thing template is defined like this:

```
Thing template 'vocabWords' 'name' @location? "desc"?
```

This means that typical Thing definitions will tend to look like this:

```
brassCoin : Thing '(small) brass coin/groat*coins' 'small brass coin' @longCave
    "On the obverse is the head of King Freddie the Fat, and on the reverse
    is stamped ONE GROAT. "
;
```

Which is exactly equivalent to:

```
brassCoin : Thing
    vocabWords = '(small) brass coin/groat*coins'
    name = 'small brass coin'
    location = longCave
    desc = "On the obverse is the head of King Freddie the Fat, and on the reverse
            is stamped ONE GROAT. "
;
```

TADS 3 Tour Guide

Or this:

```
++ fluidLink : Thing 'fluid link' 'fluid link'
    "It's a long transparent tube. Both ends are capped with some kind of shiny
    metal, and at one end is a tiny hole. "
;
```

Which is exactly equivalent to:

```
++ fluidLink : Thing
    vocabWords = 'fluid link'
    name = 'fluid link'
    desc = "It's a long transparent tube. Both ends are capped with some kind of shiny
    metal, and at one end is a tiny hole. "
;
```

The main difference is that the second example, the `fluidLink`, uses the `++` notation to specify its location relative to some previously defined object, so that it does not need to set its location property by any other means. Since `@location?` in the `Thing` template includes a question-mark to show that this element is optional, it can be omitted from the object definition and the template will still match. The `brassCoin`, however, does not use the `+` syntax to determine its location, so this needs to be done some other way; hence we specify its location using `@longCave`.

You may define the occasional `Thing` that is so insignificant that it does not merit a description, in which case you can simply omit the double-quoted string from the definition, making for extremely concise code, e.g.:

```
+ peanut: Food 'peanut/nut' 'peanut';
```

or

```
peanut : Food 'peanut/nut' 'peanut' @kitchenTable;
```

This also illustrates how subclasses of `Thing` (of which `Food` is one) can use the same template as `Thing`.

There's two further types of template we ought to consider; the first is one that can match *alternatives* at the same location within the sequence of properties. Here's a simple example from the library:

```
DefaultTopic template "topicResponse" | [eventList];
```

This template means that you can define *either*

```
DefaultTopic "Bob looks bored with your question";
```

Meaning

```
Default Topic
    topicResponse = "Bob looks bored with your question"
;
```

Or

```
DefaultTopic [ 'Bob looks bored', 'Bob yawns', 'Bob is so bored he falls asleep'];
```

Meaning

```
DefaultTopic
    eventList = [ 'Bob looks bored', 'Bob yawns', 'Bob is so bored he falls asleep']
;
```

(Which isn't actually very useful unless your `DefaultTopic` also inherits from an `EventList` class, but that's another matter).

A more complex example is provided by:

```
TopicEntry template +matchScore?
    @matchObj | [matchObj] | 'matchPattern'
    "topicResponse" | [eventList] ?;
```

TADS 3 Tour Guide

Which can be matched by something as simple as

```
TopicEntry @bob
  "<q>That's none of your business!</q> he declares. "
;
```

Or something as complex as:

```
TopicEntry + 110 [silverCoin, goldCoin, brassCoin]
[
  ' <q>I\'ve never been interested in coins.</q> he growls. ',
  '<q>Don\'t try to tempt me with money - I can\'t stand the stuff.' he complains. ',
  '<q>Filthy lucre! Take it away!</q> he demands. ',
  '<q>The root of all evil.</q> he opines '
]
;
```

Although we shan't try to run through all the possible permutations here.

The remaining type of template we need to consider is that which uses the *inherited* keyword in its definition. In fact, the library defines very few of these; one (fairly important) example is:

```
Passage template ->masterObject inherited;
```

In this context the *inherited* keyword refers to the templates of *all* Passage's superclasses, so this template could potentially represent a series of templates, in which *inherited* is replaced with the template of each of Passage's superclass in turn (and also with nothing). Passage inherits from Linkable, Fixture and TravelConnector, none of which defines a template. Linkable inherits from object (so there's no template there). Fixture inherits from NonPortable which inherits from Thing which inherits from VocabObject; TravelConnector also inherits from Thing. The possible templates Passage can inherit from are therefore those for [Thing](#) and for [VocabObject](#). This foregoing definition is thus equivalent to the following:

```
Passage template -> masterObject;
Passage template -> masterObject 'vocabWords';
Passage template -> masterObject 'vocabWords' 'name' @location? "desc"??;
```

Note that this is almost but not quite equivalent to:

```
Passage template -> masterObject 'vocabWords'? 'name'? @location? "desc"??;
```

The reason it is *not* equivalent is that this template would allow the location or desc properties to be specified in the template without the name property, which the real Passage template will not.

Note that since Passage inherits from Thing and VocabObject, it is *also* perfectly legal to use the Thing and VocabObject templates with a Passage, e.g.:

```
Passage 'passage';
Passage 'long passage' 'long passage' @diningRoom "The long passage leads into the hall. ";
```

All this actually looks a good deal more complicated than it will ever work out in practice, for in practice, if you want to use a Passage (or one of its subclasses) you will *either* use the Thing template to define it, or the form of the Passage template in which *inherited* picks up the Passage template. Thus, although you can use other template combinations with Passage, in practice most of the time (perhaps 99% of the time), you will use Passage and its subclasses as *if* its template were defined:

```
Passage ->masterObject? 'vocabWords_' 'name' @location? "desc"??;
```

This applies equally to the other classes for which the library defines templates including the inherited keyword, namely [Enterable](#) and [Exitable](#).

Finally, to see how templates work with multiple inheritance, consider the following:

```
class TestA : object
  weight = 0
  colour = nil
  mydesc = nil
;
```

TADS 3 Tour Guide

```
Class TestB : object
    bulk = 0
    texture = nil
;

class TestC : TestB, TestA;
;

TestA template +weight 'colour' 'mydesc'?;
TestB template +bulk 'texture';

testMe : TestC +20 'rough' ;

testMeAgain : TestC +30 'red' 'wooden';
```

Results in

```
testMe : TestC
    weight = 0
    colour = nil
    myDesc = nil
    bulk = 20
    texture = 'rough'
;

testMeAgain: TestC
    weight = 30
    colour = 'red'
    myDesc = 'wooden'
    bulk = 0
    texture = nil
;
```

The testMe object has a definition that in principle could match either the TestA template or the TestB template. It is the TestB template that is actually matched because TestB comes earlier in the class list of TestC. On the other hand testMeAgain has a definition that can only match the TestA template, so it is the TestA template that is matched.

Finally, we should consider how the *inherited* keyword works in the context of multiple inheritance. If we now go on to define:

```
TestC template inherited 'shape';
```

The 'inherited' keyword can inherit *any* of the templates from any of TestC's superclasses, or else nothing at all. The definition is thus equivalent to defining the following three templates:

```
TestC template 'shape';
TestC template +bulk 'texture' 'shape';
TestC template +weight 'colour' 'mydesc'? 'shape';
```

Note also that objects of class C (such as testMe and testMeAgain) will also continue to match templates defined on its superclasses (in this case, the templates for ClassA and ClassB).

Suppose we also define an object:

```
testMeShape : TestC +10 'blue' 'large' 'square';
```

Now, this can *only* match the last form of the template, so it will mean weight=10, colour='blue', mydesc='large' and shape='square'. But what of our previous two objects?

As before, testMe has bulk=20, texture='rough', while testMeAgain has bulk=30, shape='wooden', texture='red'. Since the TestC template is defined later in the file than the other two, the other two still match first.

1.4. Startup Code : gameMain

Before we can start writing the game proper, we need to provide a tiny amount of startup code. Since TADS 3.0.6n the startup code you have to supply for a game has been reduced to a minimum. Basically all you have to do is to define a `gameMain` object that specifies the player character, perhaps sets some options, and (optionally) shows the introductory and concluding messages, something like:

```
gameMain : GameMainDef
    initialPlayerChar = me
    showIntro()
    {
        "Finding yourself at a loose end in the Parser Valley,
        you have wandered up to take a look at the famous
        Eerhtsdad Caves. You're not entirely sure what they're
        famous for, or why they should be worth a look, but that's
        what it said the guidebook you found abandoned on the
        back seat of the bus, so it must be true. Anyway, you're
        here now, so you reckon you may as well take a look.\b";
    }

    setAboutBox()
    {
        "<ABOUTBOX><CENTER>The Quest of the Golden Banana\b
        v <<versionInfo.version>>\b
        (c) 2004 Eric Eve\b
        </CENTER></ABOUTBOX>";
    }

    showGoodBye()
    {
        "<.p>Thanks for playing!";
    }
;
```

You can do more than this on `gameMain`. Later on, for example, we'll be discussing how you can set up the [maximum score](#) and a [score rank table](#) here. You can also set the properties **allowYouMeMixing** (true by default), and **showExitsInStatusline** (also true by default) In case it isn't obvious what these do, here's how the comments in the library code describe them:

- **allowYouMeMixing** - Option flag: allow the player to use "you" and "me" interchangeably in referring to the player character. We set this true by default, so that the player can refer to the player character in either the first or second person, as long as the player character normally uses either or these (in other words, this option is meaningless in a game when the narration refers to the player character in the third person). If desired, the game can set this flag to nil to force the player to use the correct pronoun to refer to the player character. We set the default to allow using "you" and "me" interchangeably because this will create no confusion in most games, and because most experienced IF players will be accustomed to using "me" to refer to the player character (because the majority of IF refers to the player character in the second person, and expects the player to conflate the player character with the player and hence to refer to the player character in the first person). It is relatively unconventional for a game to refer to the player character in the first person in the narration, and thus to expect the player to use the second person to refer to the PC; as a result, experienced players might tend to use the first person out of habit in such games, and might find it jarring to find the usage unacceptable. Furthermore, in games that use a first-person narration, it seems unlikely that there will also be a second-person element to the narration; as long as both aren't present, it will cause no confusion for the game to accept either "you" or "me" as equivalent in commands. However, the library provides this option in case such a situation does arise.
- **allVerbsAllowAll** - if this option flag is set to nil, ALL (as in TAKE ALL or X ALL) will only be allowed with the basic inventory management commands TAKE, TAKE FROM, DROP, PUT IN and PUT ON. By default `allVerbsAllowAll` is true, which means that ALL can be used with all verbs that allow multiple direct objects (or multiple indirect objects if your game defines any such verbs). If you wish, you can also override the **actionAllowsAll** property on individual actions to determine which of them will and will not accept ALL as a noun phrase.

TADS 3 Tour Guide

- **initialPlayerChar** - The initial player character. Each game's 'gameMain' object MUST define this to refer to the Actor object that serves as the initial player character.
- **showExitsInStatusline** - Flag: show automatic exit listings in the status line. We enable this by default. This is an author-configured option. The library doesn't provide a command to let the player control this setting (although a game could certainly add one, if desired).
- **usePastTense** - Flag: if true, the game will be narrated in the [past tense](#) instead of the present tense (e.g. "On the table was a banana" instead of "On the table is a banana"). This flag can also be switched in-game to switch between past-tense and present-tense narration.
- **verboseMode** - Prior to version 3.0.9 this was a logical (true/nil) flag; if it was true, the full room description was displayed each time the player enters a room, regardless of whether or not the player has seen the room before; if nil, the full description is only displayed on the player's first entry to a room, and only the short description on re-entry. Note that the library provides VERBOSE and TERSE commands that let the player change this setting dynamically. From TADS 3.0.9 this property has become a BinarySettingsItem that shouldn't be overridden by the game author, first because doing so will almost certainly cause a run-time error, and second because the intention with the mechanism introduced in version 3.0.9 is that it is up to players rather than authors to set the default they require. The moral: leave this property alone unless you're *very* sure what you're doing and have a very good reason for doing it. Moreover, if you're upgrading an existing game from a pre-3.0.9 version to 3.0.9 or later, make sure your gameMain *doesn't* override this property.

Our gameMain object has defined the player character as an object called me, so we next need to define this object, which, at a minimum, means assigning it to an appropriate class and providing it with an initial location:

```
me: Actor
  desc = "You look even better than the last time you looked. "
  /* the initial location */
  location = outsideCave
;
```

We'll get round to defining the [outsideCave](#) location shortly. In the meantime there's one more job we might want to get out of the way at this stage, and that is to define the versionInfo object, which provides important information about the game:

```
versionInfo: GameID
  IFID = 'cd03d4a8-f39b-ae69-693d-5fddc65f6dd8'
  name = 'The Quest of the Golden Banana'
  byline = 'by Eric Eve'
  htmlByline = 'by <a href="eric.eve@hmc.ox.ac.uk">
    Eric Eve</a>'
  version = '1.0'
  authorEmail = 'Eric Eve <eric.eve@hmc.ox.ac.uk>'
  desc = 'A combination of cave exploration and time-travel with clear
    allusions both to the Lord of the Rings and Dr Who, this game is
    primarily an example game to provide a tutorial on the adv3
    library for aspiring TADS 3 game authors.'
  htmlDesc = 'A combination of cave exploration and time-travel with clear
    allusions both to <i>The Lord of the Rings</i> and <i>Dr Who</i>, this game is
    primarily an example game to provide a tutorial on the adv3
    library for aspiring TADS 3 game authors.'

  showCredit()
  {
    /* show our credits */
    "TADS 3 language and library by Michael J. Roberts ";

    /*
     * The game credits are displayed first, but the library will
     * display additional credits for library modules. It's a good
     * idea to show a blank line after the game credits to separate
     * them visually from the (usually one-liner) library credits
     * that follow.
     */
    "\b";
  }
  showAbout()
  {
    "Although this game is winnable, and some players may find it
```


TADS 3 Tour Guide

tolerably entertaining, it is primarily designed as a sample game and programming exercise to accompany the <i>TADS 3 Tour Guide</i>. The game has thus been designed to give authors a reasonably comprehensive tour of the library, rather than as a satisfactory playing experience by the standards of modern IF. This may result in (a) a certain quirkiness about the whole game, (b) somewhat bizarre and derivative plotting and (c) incomplete implementation of non-essential aspects of the game such as hints, scoring, and decoration objects. This is because the game's primary audience - people trying to learn how to program with the TADS 3 library - only need a limited number of examples of each feature.\b There should, however, be no actual bugs in the game (that is, there are not <i>meant</i> to be any actual bugs, although there almost certainly <i>will</i> be some in practice), so should you encounter any, the author would be grateful for a bug report. ";

The first half of this object definition basically defines the bibliographical metadata for the game (for a full explanation see the 'Bibliographical Metadata' article in the *Technical Manual*). Note in particular the first field, **IFID**. This is a unique identifier for your game (a little like an ISBN number for published books), which must be unique to your game. It is essentially a set of random hexadecimal digits (0-9, a-f) in the format `xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx`. If you create a project in Workbench this number will automatically be generated for you. If you are creating your project by some other means, you will need to ensure that you add such a number to your definition of `versionInfo`. To obtain an IFID that is guaranteed to be truly random, as this needs to be (to ensure the avoidance with IFID numbers assigned to other games), you can use the TADS IFID generator at <http://www.tads.org/ifidgen/ifidgen>. The other bibliographical data (such as the name of the game and its author) should be fairly self-explanatory (but see the 'Bibliographical Metadata' article for full details). The final two methods contain the text that should be displayed in response to the **credits** and **about** command. Of course, you may prefer the latter to launch a menu rather than just display a text dump.

2. Rooms and Connectors

2.1. Introduction

In the sections that follow we shall endeavour to make use of all the main types of room and travel connector in the TADS 3 library.

Rooms are locations in which actors and other objects may exist, and between which actors may travel. Since travel is possible directly from one Room to another, Rooms are also Travel Connectors. TravelConnectors allow travel between Rooms: their class hierarchy is

```

TravelConnector
  Passage
    Stairway
      StairwayDown
      StairwayUp

    ThroughPassage
      Door
        AutoClosingDoor
      ExitOnlyPassage
      PathPassage
      SecretDoor
      HiddenDoor

  RoomConnector
    OneWayRoomConnector
    RoomAutoConnector
    Room
      DarkRoom
      FloorlessRoom
      OutdoorRoom

  TravelMessage
    NoTravelMessage
    FakeConnector

AskConnector

```

Note that Passage also descends from Fixture, so that Passage and all its subclasses represent physical game objects as well as connectors. This is not the case with RoomConnector and its descendants or TravelMessage and its.

Note that TravelMessage also descends from [TravelWithMessage](#).

There is also a [ShipBoardRoom](#) class that can be used as a mix-in class for other kinds of room.

Room and its subclasses have a number of methods and properties that it is sometimes useful to override, these include:

```

atmosphereList
brightness
destName
enteringRoom
roomAfterAction
roomBeforeAction
roomParts

```

2.2. OutdoorRoom

We'll start our adventure outside a cave, so we'll begin by defining our first room thus:

```
outsideCave : OutdoorRoom 'Parser Valley' 'Parser Valley'
    "To the north stretches the broad green Parser Valley under a clear blue sky,
    past a small car park lying just off to the east. The main feature of
    interest round here is the notorious Eerhtsdats Caves, the entrance to which
    lies just to the south, marked by a large blue sign that proclaims, predictably
    enough:\b<FONT FACE='TADS-Typewriter' BGCOLOR=BLUE COLOR=WHITE>
    ENTRANCE TO THE\neERHTSDAT CAVES</FONT>\n"

    atmosphereList : ShuffledEventList {
    [
        '\nA flight of birds disappears off to the west. ',
        { : "\nA <<rand('small', 'large')>> <<rand('green', 'red', 'blue', 'black', 'white')>> car
          pulls out of the car park and drives off to the north. " },
        '\nAn aeroplane flies far overhead. ',
        nil
    ]
    }
;
```

We use the class `OutdoorRoom` for the obvious reason that it represents an outdoor location (with no walls, and with ground and sky rather than floor and ceiling). Recall that we have already set the location property of the `me` object to `outsideCave` so that the player character will begin here.

Remember that the [Room template](#) (which also applies to `OutdoorRoom`) allows this abbreviated form of definition: the template is defined as `'roomName' 'destName'? 'name'? "desc"?`; which means that the first single-quoted string after the class name is the `roomName` property (the name that will be shown in the status line for the room), the second (which is optional) the `destName` (the name by which the room will be referred to in an exit lister) and the double-quoted string is the `desc` property (which will be displayed as the room description). If all these properties are used, they must be used in the order defined by the template, and before any other properties are defined for the room.

In other words, the definition:

```
outsideCave : OutdoorRoom 'Parser Valley' 'Parser Valley'
    "To the north stretches the broad green Parser Valley under a clear blue sky,
    past a small car park lying just off to the east. The main feature of
    interest round here is the notorious Eerhtsdats Caves, the entrance to which
    lies just to the south, marked by a large blue sign that proclaims, predictably
    enough:\b<FONT FACE='TADS-Typewriter' BGCOLOR=BLUE COLOR=WHITE>
    ENTRANCE TO THE\neERHTSDAT CAVES</FONT>\n"
;
```

is exactly equivalent to writing out in full:

```
outsideCave : OutdoorRoom
    roomName = 'Parser Valley'
    destName = 'Parser Valley'
    desc = "To the north stretches the broad green Parser Valley under a clear blue sky,
    past a small car park lying just off to the east. The main feature of
    interest round here is the notorious Eerhtsdats Caves, the entrance to which
    lies just to the south, marked by a large blue sign that proclaims, predictably
    enough:\b<FONT FACE='TADS-Typewriter' BGCOLOR=BLUE COLOR=WHITE>
    ENTRANCE TO THE\neERHTSDAT CAVES</FONT>\n"
;
```

At this point we should pause to consider the relation between some of these properties. The `roomName` is the room title displayed in the room description and the status line; typically, this will be in title case (e.g. "Hall of the Mountain King"). The `destName` is the title given to the room in the exit lister that appears in response to the 'exits' command, or when you try to move in direction you can't go (e.g. "north, back to the hall of the mountain king"). The plain name property is the title used by the parser to refer to the room when it features in commands (which can normally only occur if the room is given `vocabWords`). By default `name` is defined as `roomName.toLower`, and `destName` is defined as `theName`. Often this gives reasonable results, but you might often want to override it, as in this case where `Parser Valley` is a proper name we want used both for the `roomName` and the `destName`.

TADS 3 Tour Guide

We'll define one extra property for `OutsideRoom` at this point, namely its `atmosphereList`. If this is defined to hold a [Script](#) object, the `roomDaemon` will automatically call its `doScript` method each turn; in practice this means we can make it an anonymous nested object of a `Script` class. Here we use a [ShuffledEventList](#) to display a series of strings in random order.

In order to vary the description of cars leaving the car park, we use the `rand()` function to choose both the size and the colour of the car. When the `rand` function contains a list, it returns one item in the list chosen (notionally) at random. In order to include the two uses of the `rand` function in the string, we have made it a double-quoted string using the `<<>>` syntax. A double quoted string cannot be used as an element in the `eventList` property of an `ShuffledEventList`, but an anonymous function can, and the double-quoted string can be printed within the anonymous function. When the anonymous function consists of only a single statement, as here, we can use the short form syntax shown, i.e.

```
{: statement }
```

Note that the statement should then *not* be concluded with a semicolon.

2.3. FakeConnector

The room [outsideCave](#) was defined previously. Its description refers to a valley to the north and a car park to the east. We do not want the Player Character to go wandering off in those directions, but there should be a reasonable response to any attempts to do so; in particular the game should respond with a sensible message if the player types the commands `EAST` or `NORTH`. The `FakeConnector` is just the job for this sort of situation, where we want to provide a soft boundary. The two `FakeConnectors` to be added to the room definition are shown in bold.

```
outsideCave : OutdoorRoom 'Parser Valley' 'Parser Valley'
  "To the north stretches the broad green Parser Valley under a clear blue sky,
  past a small car park lying just off to the east. The main feature of
  interest round here is the notorious Eerhtsdats Caves, the entrance to which
  lies just to the south, marked by a large blue sign that proclaims, predictably
  enough:\b<FONT FACE='TADS-Typewriter' BGCOLOR=BLUE COLOR=WHITE>
  ENTRANCE TO THE\neERHTSDAT CAVES</FONT>\n"
  north : FakeConnector { "You've come here to explore the caves, not the valley. " }
  east : FakeConnector { "You've only just come from there -- you've no reason to go back just
now. " }
  atmosphereList : ShuffledEventList {
    [
      'A flight of birds disappears off to the west. ',
      {:"A <<rand('small', 'large')>> <<rand('green', 'red', 'blue', 'black', 'white')>> car
      pulls out of the car park and drives off to the north. " },
      'An aeroplane flies far overhead. ',
      nil
    ]
  }
;
```

At this point you can compile and run the game to test that it is working properly.

Note that we once again use a template to abbreviate the business of writing the `FakeConnector` definition. The property in double quotes that we are defining for each `FakeConnector` here is in fact its `travelDesc` property (defined by the [NoTravelMessage template](#), which `FakeConnector` inherits). This is called by the connector's `showTravelDesc()` method only for the Player Character (so that, for example, the message will not be shown repeatedly if the PC is being accompanied by one or more NPCs), while `showTravelDesc()` is in turn invoked by `noteTraversal(traveler)`. The last of these methods - `noteTraversal` - is defined for all `TravelConnectors`, whereas the other two - `travelDesc` and `showTravelDesc` - are defined on [TravelWithMessage](#) and classes that descend from it.

The `FakeConnector` works very like the [NoTravelMessage](#). The only difference is that a direction attached to a `NoTravelMessage` won't be included in a list of exits (e.g. in response to an `EXITS` command, or in the status line), whereas that attached to a `FakeConnector` will. A `NoTravelMessage` should therefore be used to explain why travel is not possible in a direction in which it's reasonably apparent that travel isn't possible, while a `FakeConnector` should be used to make travel apparently possible in a direction in which it isn't really, e.g.. to provide a "soft boundary" to the map.

2.4. DeadEndConnector

In the previous section we added a pair of FakeConnectors to prevent the player character from going wandering north into the valley or east into the car park, although there's nothing physically preventing him from doing so. When using the FakeConnectors for this purpose we basically blocked the PC from travelling north or east by providing him motivational reasons for not doing so. The alternative would be allow him to do so, but then have him return to his starting point (either because the way turns out to be blocked, or because the PC finds nothing of interest). So instead of the FakeConnectors used in the previous section, we could use a pair of DeadEndConnectors thus:

```
outsideCave : OutdoorRoom 'Parser Valley' 'Parser Valley'
  "To the north stretches the broad green Parser Valley under a clear blue sky,
  past a small car park lying just off to the east. The main feature of
  interest round here is the notorious Eerhtsdats Caves, the entrance to which
  lies just to the south, marked by a large blue sign that proclaims, predictably
  enough:\b<FONT FACE='TADS-Typewriter' BGCOLOR=BLUE COLOR=WHITE>
  ENTRANCE TO THE\neERHTSDAT CAVES</FONT>\n"
  north : DeadEndConnector { 'Parser Valley'
    "You start to stride off into the valley, but soon decide it's not that interesting,
    so you wander back towards the cave entrance. " }
  east : DeadEndConnector { 'the car park'
    "You go and wander round the car park for a few minutes, but decide you don't want to
    leave just yet, so you return to the cave entrance. " }

  atmosphereList : ShuffledEventList {
  [
    'A flight of birds disappears off to the west. ',
    { : "A <<rand('small', 'large')>> <<rand('green', 'red', 'blue', 'black', 'white')>> car
      pulls out of the car park and drives off to the north. " },
    'An aeroplane flies far overhead. ',
    nil
  ]
  }
;
```

At first sight it may look as if we could have used a FakeConnector for this purpose and it would have done the job just as well, and this is indeed almost the case. Nevertheless there are a couple of distinctions between FakeConnector and DeadEndConnector that are worth observing, even if they may seem a bit subtle at first sight.

The first is that traveling via a DeadEndConnector triggers travel notifications while attempting to travel via a FakeConnector does not. So, for example, suppose there was an NPC present who might react to our attempts to walk away from the cave entrance; suppose that if we try to go in any direction except south into the cave she (assuming a female NPC) objects and prevents our leaving (we'd implement this with a beforeTravel() method on the NPC's current ActorState, but that's the sort of thing we'll be coming to some way ahead, so we shan't worry about the details just now). If we used a FakeConnector to represent what happens when the PC tries to go north or east, then we'd never see the NPC's protest. If we used a DeadEndConnector, however, the NPC's protest would be triggered, and we'd see her protest in place of the message describing our wandering round the valley or car park. The first case, using a FakeConnector, is appropriate in situations where the PC doesn't even attempt to travel and we're simply displaying a message explaining why not; since the PC doesn't attempt to travel, there's no reason why anyone or anything should react to his non-attempt. The second case is appropriate when the PC does (at least notionally) attempt the travel, and where the message we display describes that (albeit simulated and circular) travel unless something or someone acts to prevent it, such as our (for now) putative female companion who insists on our entering the cave instead.

So, in brief:

- Use a *FakeConnector* to explain why your PC refuses to attempt travel in a direction in which travel would be physically possible.
- Use a *DeadEndConnector* to simulate the effect of your PC travelling in a direction (which doesn't actually connect to another location on your game map) and then returning to his starting point.

And now on to the second difference. If you look at the code we just changed, you'll see that we added a second property in the DeadEndConnectors, just before the double-quoted strings describing the aborted walk into the valley

TADS 3 Tour Guide

and car park. These extra properties are the single-quoted strings 'Parser Valley' and 'the car park', which name the locations to which these connectors notionally lead (although in reality they lead nowhere and we aren't going to implement a Parser Valley or car park location in our game). The property to which we are giving a value here is called **apparentDestName**; the point of it is that the exit lister (shown in response to an explicit EXITS command or an attempt to move in a direction for which no connector has been defined) will show these as the destinations that can (notionally) be reached by travelling via the DeadEndConnector. For example, an EXITS command issued in our starting location might generate the response:

Obvious exits lead north to Parser Valley, south, and east to the car park.

If you compile the game and try it out as it stands, however, you'll find these destination names appear only after the PC has attempted to travel via these DeadEndConnectors. In some situations (namely where the PC doesn't know where a connector leads till he tries traversing it) this may be just what we want. In this case, however, it's perfectly obvious from where the PC's standing that the valley is to the north and the car park to the east, so ideally we'd like these destination names to appear even before the PC attempts to travel. We can do this by overriding the **actorKnowsDestination** method on the location to indicate which connectors the PC already knows the destinations of even without travelling:

```
outsideCave : OutdoorRoom 'Parser Valley' 'Parser Valley'
...
actorKnowsDestination(actor, conn)
{
    return conn is in (east, north) ? true : inherited(actor, conn);
}
;
```

There are two further points to note about this. In the above method east and north are actually references to our two DeadEndConnectors. Neither DeadEndConnector has a name of its own, so the only way of referring to them is via the properties to which they are attached, namely outsideCave.north and outsideCave.east. Since, in this case, we are referencing these properties from a method of outsideCave, we don't need to prepend the object name to them; in this context they can be referred to simply as 'east' and 'north' meaning the TravelConnectors attached to the east and north properties of the current object.

The second point is that we're not restricted to using actorKnowsDestination with DeadEndConnectors; the method can be used to signal that the NPC already knows the destination of *any* kind of TravelConnector (including another Room, if a direction property points straight to another Room, as is usually most often the case).

2.5. RoomConnector

So far we can't actually leave the starting location. We could simply define the next location and simply point to it from the starting room, but in this case we want to make the tunnel into the cave subject to a rockfall that may block it (in either direction). Once the player starts exploring the cave system, he or she will then have to find another way out.

An efficient way to perform this task is with a RoomConnector, since we can conditionally block passage through it. We can define the appropriate RoomConnector thus:

```
entranceTunnel : RoomConnector
    room1 = entranceCave
    room2 = outsideCave
    blocked = nil
    canTravelerPass (traveler) { return !blocked; }
    explainTravelBarrier (traveler)
    {
        "After a few paces down the tunnel it becomes all too clear
        that it has been blocked by a recent rockfall, so there is
        nothing for it but to turn round and go back. ";
    }
;
```

The properties room1 and room2 define the two rooms that will be linked by this connector (note that we haven't defined [entranceCave](#) as yet, so the game won't compile till we do). We define a custom `blocked` property to determine whether or not the tunnel has been blocked by the rockfall. The **canTravelerPass** method (defined on all TravelConnectors) determines whether a traveler can traverse this connector. In this case we want to allow travelers

TADS 3 Tour Guide

to pass if the connector is not blocked, but not otherwise, so we simply returned `!blocked` (i.e. not blocked). If travel is forbidden the **`explainTravelBarrier`** method is invoked, so we define it to display an appropriate message in the event that the tunnel is blocked.

Note that the tunnel is not represented as a physical object in the game (although it could have been): the `RoomConnector` is an *abstract* object linking the two rooms (although in a sense it does duty for a representation of a tunnel that can be blocked).

Note also that it will be necessary to make the appropriate direction properties of both [outsideCave](#) and [entranceCave](#) point to this `RoomConnector`. We'll do that next.

2.6. asExit

The `asExit()` macro can be used when we want more than one direction to point to the same destination, but we only want one of the directions to appear in the list of exits (the others effectively being synonyms). In the `outsideCave` room the cave entrance is described as lying to the south, so that the Player might type either `SOUTH` or `IN` to enter it. Here we'll make `SOUTH` the explicit way in and add handling for `IN` as a synonym using `asExit`:

```
outsideCave : OutdoorRoom 'Parser Valley' 'Parser Valley'
  "To the north stretches the broad green Parser Valley under a clear blue sky,
  past a small car park lying just off to the east. The main feature of
  interest round here is the notorious Eerhtsdats Caves, the entrance to which
  lies just to the south, marked by a large blue sign that proclaims, predictably
  enough:\b<FONT FACE='TADS-Typewriter' BGCOLOR=BLUE COLOR=WHITE>
  ENTRANCE TO THE\neERHTSDAT CAVES</FONT>\n"
  north : FakeConnector { "You start to stride off into the valley, but soon
    decide it's not that interesting, so you wander back towards the cave
    entrance. " }
  south = entranceTunnel
  in asExit(south)
  east : FakeConnector { "You go and wander round the car park for a few
    minutes, but decide you don't want to leave just yet, so you return
    to the cave entrance. " }
  atmosphereList : ShuffledEventList {
  [
    'A flight of birds disappears off to the west. ',
    { : "A <<rand('small', 'large')>> <<rand('green', 'red', 'blue', 'black', 'white')>> car
      pulls out of the car park and drives off to the north. " },
    'An aeroplane flies far overhead. ',
    nil
  ]
  }
};
```

Once again, the new properties to be added are shown in bold. Note that we point the south property not to another room, but to the previously defined [RoomConnector](#), `entranceTunnel`.

2.7. Enterable

The room definition for [outsideCave](#) will work fine (once we have defined the [entranceCave](#) Room) if the player types `IN` or `SOUTH` or even `ENTER`, but since the room description mentions a cave, the player may try to `ENTER CAVE` or `EXAMINE CAVE`. To cover this possibility we should define a cave object and make it enterable:

```
+ Enterable ->entranceTunnel 'eerhtsdats cave/entrance/caves' 'cave'
  "The entrance to the cave is large and welcoming; two large people could easily walk in
  side by side without stooping. "
;
```

This definition uses the [Enterable template](#).

TADS 3 Tour Guide

We use the + syntax to locate this Enterable in outsideCave (so make sure its definition comes after that of outsideCave and before anything else).

Since there is no need to refer to this object from anywhere else in our game code we can define it as an anonymous object; there is no need to give it an object name, we simply use the class name (without a preceding colon).

Important Note.

Enterable, in common with [EntryPortal](#), [Exitable](#) and [ExitPortal](#), superficially resembles Passage-type objects like [ThroughPassage](#) in that it represents a game object which one can go through and end up in a different location. Unlike the various Passage objects (ThroughPassage, Door) etc., Enterable, Exitable, EntryPortal and ExitPortal are *not* TravelConnectors and have none of the TravelConnector methods or properties. Also, unlike Passages and Doors, they do not descend from Linkable, which means, for example, that an EntryPortal cannot be the masterObject of an ExitPortal; these classes cannot be linked together as pairs pointing to each other.

Passages and Doors typically refer to other destinations through their **otherSide** or **destination** properties.

Enterable, EntryPortal, Exitable and ExitPortal refer to their destination through their **connector** property, which may simply be set to the location you want an actor to end up in when entering or exiting such an object, but may instead be set to a TravelConnector object.

2.8. Room

We can now define the second room in the game. Since this will be an interior room (albeit inside a cave rather than a building) we'll make it of the Room class:

```
entranceCave : Room 'Entrance Cave' 'the entrance cave'
    "Compared with the narrow tunnel leading out to the north, this
    rough-walled cave seems positively spacious. A red sign fixed to
    one wall suggests that the narrow tunnel is the only way back out to
    the valley, while a blue sign next to it welcomes you to the cave.
    Directly under the signs a narrow ledge has been carved into the
    wall. There appear to be no other caves at this level, but a sturdy
    steel ladder leads down through a large round hole in the floor. "

    north = entranceTunnel
    out asExit(north)
;
```

Note that the cave's north property points to the previously defined [RoomConnector](#), and that we use the [asExit](#) macro to allow OUT as a synonym for NORTH.

Once again, note the use of the [Room template](#) to define the common properties of this Room. The first single-quoted string, 'Entrance Cave' is the name of the Room. The second 'the entrance cave' (which is optional - we could just leave it out) is its destName (the name that will appear in exit listings). The double quoted string that follows, "This large cave..." is the room description.

Although this is an underground cave, we assume it will be permanently lit by some means or other. In more complex situations you might want to override the brightness property to vary according to circumstance (as is exemplified in the definition of the [secretPassage](#), which comes later).

At this point it should be possible to compile and test the game once more.

2.9. StairwayDown

The description of `entranceCave` refers to a sturdy steel ladder leading down through a hole in the floor. This ladder is best implemented as a `stairwayDown`, which is both a physical game object that can be examined and a `TravelConnector` that can be traversed, by `CLIMB` and `CLIMB DOWN` commands. The ladder can simply be defined as:

```
+ downLadder : StairwayDown 'sturdy steel ladder' 'sturdy steel ladder'
  "The ladder leads down through a large hole in the floor. "
;
```

Here we are simply using the standard [Thing template](#), although since `StairwayDown` inherits (indirectly) from `Passage`, it can also use the [Passage template](#).

We can then add a `down` property to the room definition to point to this connector:

```
entranceCave : Room 'Entrance Cave' 'the entrance cave'
  "This is the main cave. A large rock rests against the north wall and
  there are other caves to south and east, but the way west is blocked by
  a huge boulder. A blazing torch is fixed to the wall, next to a sturdy
  steel ladder leading upwards. "
  north = entranceTunnel
  out asExit(north)
  down = downLadder
;
```

Note that as yet nothing defines where we end up when we go down the ladder. This is because there will be a corresponding [StairwayUp](#) in the cave below, and the `StairwayUp` will point to `downLadder` as its **masterObject**. The game will automatically link the `StairwayUp` to its `masterObject` and *vice versa*, so that when we traverse the `StairwayDown` it will know that its destination is in the corresponding `StairwayUp`'s location. (We could equally well do this the other way round and make the `StairwayUp` the `masterObject` of the `StairwayDown`).

2.10. StairwayUp

We first need to add a minimal definition of the room in which we'll put the bottom end of the ladder:

```
mainCave: Room 'Large Cave'
  "The flickering orange light from the blazing torch fixed to the wall
  accentuates the naturally ruddy hues of this large, irregular cave,
  which seems to be something of a major hub in the cave system. A
  large rock rests against the wall to the north.
  A sturdy steel ladder leading upwards. "

  up = upLadder
;
```

The main thing to note here is that we point the `up` property of the room to the `upLadder` object we're about to define, so that it can be traversed either in response to an `UP` command, or in response to a `CLIMB (UP) LADDER` command. We next define the basic `upLadder` object (using the [Passage template](#)):

```
+ upLadder : StairwayUp ->downLadder
  'sturdy steel ladder' 'sturdy steel ladder'
  "The ladder leads up through a hole in the ceiling. "
;
```

The one thing to note here is the use of the `->` in the template syntax to link the `upLadder` to its `masterObject`, the corresponding [StairwayDown](#), `downLadder`. The two `Stairway` objects are now linked so that traversing one will take us to the location of the other (we could equally well have done this the other way round by having `downLadder` point to `upLadder` as its `master object`, although we would *not* want both of them pointing to each other).

Either way, our ladder will work fine, but now we want to add a refinement. Remember when we defined the `entranceTunnel` [RoomConnector](#) we gave it a **blocked** property to simulate the effect of a rockfall? Well, now we want

TADS 3 Tour Guide

to trigger the rockfall the first time the PC climbs the ladder back to the entranceCave. We could do this by overriding the stairwayUp's noteTraversal method, perhaps along the following lines (using an additional **climbed** property we define to make sure that the rockfall occurs only once):

```
+ upLadder : StairwayUp ->downLadder
  'sturdy steel ladder' 'sturdy steel ladder'
  "The ladder leads up through a hole in the ceiling. "
  noteTraversal(traveler)
  {
    if(!climbed)
    {
      "As you climb the ladder you hear what sounds like a thunderous rockfall
      up above. ";
      entranceTunnel.blocked = true;
      climbed = true;
    }
  }
  climbed = nil
;
```

There is no reason why we should not do it this way, but since we want to explore as much of the library as possible, we'll next look at another way of doing it using [TravelWithMessage](#).

2.11. TravelWithMessage

TravelWithMessage is a mix-in class for use with TravelConnectors (note that some descendents of TravelConnector - [TravelMessage](#), [NoTravelMessage](#) and [FakeConnector](#) - include TravelWithMessage in their definition in any case). TravelConnector overrides noteTraversal(traveler) to call showTravelDesc(), which in turn calls either travelDesc (if the Player Character is doing the traveling) or npcTravelDesc (if an NPC is doing the traveling).

Firstly, we'll add TravelWithMessage to the upLadder's class list so that we can use its travelDesc property. We take advantage of the fact that this will call upLadder's doScript method provided that it also inherits from the [Script](#) class or one of its descendents. In this case we'll use the [StopEventList](#) class with two items in its eventList. The first time the PC traverses the upLadder the first event in the eventList will be fired, and thereafter the second one will (defining with the [Passage template](#)):

```
+ upLadder : TravelWithMessage, StairwayUp, StopEventList ->downLadder
  'sturdy steel ladder' 'sturdy steel ladder'
  "The ladder leads up through a hole in the ceiling. "
  eventList =
  [
    new function
    {
      "As you climb the ladder you hear what sounds like a thunderous rockfall
      up above. ";
      entranceTunnel.blocked = true;
    },
    'You climb the ladder again. '
  ]
;
```

This takes advantage of the fact that an eventList can contain, inter alia, single-quoted strings (such as 'You climb the ladder again. '), which will just be displayed, or anonymous function pointers, in which case the anonymous function will be executed. To create an anonymous function containing more than one statement, as we wish to do here, we have to use the new function syntax:

```
new function
{
  statement1;
  statement2;
  ...
}
```

TADS 3 Tour Guide

In this case the function simply prints an appropriate message about the rockfall and sets [entranceTunnel](#)'s blocked property to true.

You can now recompile and test the game so far.

2.12. SecretDoor

The description of [mainCave](#) includes a rock to the north. We'll make this a secret door that reveals a secret passage behind when it is pushed to one side (using the [Thing template](#)):

```
+ rock: SecretDoor 'large rock' 'rock'
  "A large rock <<isOpen ? 'lies to one side of a passage beyond'
    : 'leans against the north wall of the cave'>> . "
  dobjFor(Push)
  {
    verify() {}
    action()
    {
      makeOpen(!isOpen);
      "The rock rolls aside. ";
    }
  }
;
```

Note that this needs to be defined just after `mainCave`, so that it is included in `mainCave`'s contents. Note also that we need to add the following to the definition of `mainCave`:

```
north = rock
```

The passage is opened by pushing the rock to one side, so we override the `action()` part of `dobjFor(Push)` to bring about the desired behaviour. `SecretDoor` descends from `BasicDoor`, which defines `makeOpen(stat)` method; this method sets the `isOpen` property to `stat`, which should be either true (for open) or nil (for closed). To make pushing the rock open the passage if it is closed, and close it if it is open, we call `makeOpen(!isOpen)`. We also test the `isOpen` property to provide a description of the rock that depends on its position.

We next need to define the location on the far side of the rock:

```
secretPassage : Room 'Secret Passage' 'the secret passage'
  "This hitherto secret passage narrows to a long tunnel running north. To the
  south <<rock2.isOpen ? 'an opening leads out into a large, ruddy-hued cave'
    : 'a large rock blocks the way out'>>. "
  south = rock2
  north = tunnel
  brightness = (rock2.isOpen ? 3 : 0)
;

/* This rock is simply the other side of the rock defined in mainCave
 * In this definition we use the Passage template
 */

+ rock2 : SecretDoor -> rock 'large rock' 'large rock'
  "It's a large rock, too heavy to lift. "
  dobjFor(Push)
  {
    verify() {}
    action()
    {
      makeOpen(!isOpen);
      "The rock rolls aside. ";
    }
  }
;
```

The second rock (`rock2`) is simply the first rock seen from the other side; we link it to the rock with `->` which defines the

TADS 3 Tour Guide

masterObject property. Otherwise everything behaves much the same as the rock, except that for variety we vary the description of rock2 in the room description.

A further refinement we can make is to have the illumination of the secretPassage room depend on the boulder's being pushed aside. If light enters the passage only through the exit into the startCave, then pushing the boulder shut while inside the secretPassage will cut off the light. To accomplish this we override the `brightness` property of secretPassage to vary according to whether the boulder is open or not.

2.13. ThroughPassage

The secretPassage room refers to a tunnel leading north. The tunnel itself isn't an interesting location, it's simply a route for getting elsewhere. The player may however try to refer to it, so we can usefully implement it as a ThroughPassage - something that you can ENTER or GO THROUGH and that takes you directly to its destination. This time we shan't implement a corresponding Passage object at the other end, so we need to set the tunnel's destination property to the room where we'll end up if we traverse the tunnel, the yet-to-be defined [smallCave](#) room.

Since we envisage this as quite a long tunnel, however, we could display a message representing the long walk down it when we travel through it; this could be achieved simply by overriding `noteTraversal`, but instead we'll take the opportunity to illustrate a simpler use of `TravelWithMessage`. All we need to do is to override `travelDesc` with the message we want displayed.

```
+ tunnel : TravelWithMessage, ThroughPassage 'tunnel' 'tunnel'
  "The dark tunnel looks large enough for a single person to
  walk through. "
  travelDesc = "You walk down the tunnel for some way and finally
  arrive in a small cave. "
  destination = smallCave
;
```

Since the tunnel is described as running north from the secretPassage, the player may simply type N or NORTH to enter it, so we need to add the following to the definition of secretRoom:

```
north = tunnel
```

We could also use a couple of anonymous ThroughPassage objects to represent the tunnel and the hole that are mentioned in the description of the entranceCave. At first sight there may seem to be a problem with this: we don't want GO THROUGH TUNNEL to bypass the RoomConnector we've set up for returning to the valley, and we'd probably want GO THROUGH HOLE to be equivalent to CLIMB DOWN LADDER. The easiest answer here is probably to remap the `TravelVia` of both ThroughPassages to the connectors we actually want employed:

```
+ ThroughPassage 'large hole' 'large hole'
  "The hole is easily large enough for even a portly giant to pass through.
  Looking through it you can see a large, rough cave below, lit by the flickering
  flames of a torch. "
  dobjFor(LookThrough) asDobjFor(Examine)
  dobjFor(TravelVia) remapTo(TravelVia, downLadder)
;

+ ThroughPassage 'narrow tunnel' 'narrow tunnel'
  "The tunnel evidently tapers from the outside to the inside, since
  the end of the tunnel visible from here is quite narrow. "
  dobjFor(TravelVia) remapTo(TravelVia, entranceTunnel);
;
```

Obviously, you should make sure that both these objects are located in entranceCave.

2.14. DarkRoom

Since by the time we end up at the end of the tunnel north from the secret passage we're now some way from the well-lit mainCave, it would not be surprising if we were now totally in the dark. We could simply override the brightness property to be 0 in the smallCave, but instead we'll make it a DarkRoom, which does this for us (using the [Room template](#)):

```
smallCave : DarkRoom 'Small Cave' 'the small cave'
    "The long narrow tunnel from the south comes to an end in this cramped,
    sandy-floored cave, whose rough rocky walls press in claustrophically
    on every side. Anyone much taller than average would have to stoop here. "
;
```

You can now compile the program and test it, but you'll quickly find that not only is there no way out of smallCave, but there's as yet no way of bringing any light to it. While developing a game it would obviously be useful to be able to test dark locations without necessarily having to use the methods the player will be obliged to use (either because you simply haven't implemented them yet, or because you don't want to have to go through the business of procuring the light source each time you want to test a new dark location). What would be useful is some way of producing light on demand while testing, and the way to do that is to provide a means of adjusting the brightness property of the Player Character object (i.e. allow the PC to be its own light source, so that it does not need to carry one). You could download Nikos Chantziaras's ncDebugActions.t library extension and use that, since it provides a number of useful debugging verbs, including MEGA and UNMEGA which (amongst other things) turns the player into a light source and back again. If for any reason you have any difficulty in acquiring this file, (which you should be able to obtain from the if-archive at <http://www.ifarchive.org/indexes/if-archiveXprogrammingXtads3XlibraryXcontributions.html>) you can get a similar effect by including the following in your own code, perhaps out of the way at the end of the file:

```
#ifdef __DEBUG

DefineIAction(FiatLux)
    execAction
    {
        if(gPlayerChar.brightness == 0)
        {
            "You start to glow!\n";
            gPlayerChar.brightness = 3;
        }
        else
        {
            "Repeating the spell reverses its effect, and your glowing aura disappears. ";
            gPlayerChar.brightness = 0;
        }
    }
;

VerbRule(FiatLux)
    'fiat' 'lux'
    : FiatLuxAction
    verbPhrase = 'make/making light'
;

#endif
```

You don't have to call it Fiat Lux, of course, you can call it anything convenient you like, but whatever you call it it's worth enclosing it between #ifdef __DEBUG (note the double underscore before DEBUG) and #endif so that this cheating verb won't be available in the release version of your game. In the debug version, however, you'll be able to type FIAT LUX (or whatever you define the command to be) to make the player character a light source, and the same command again to reverse the spell.

2.15. TravelMessage

Up to this point, you can get into the small cave but not out of it again. This time we won't explicitly mention the tunnel in the room description or implement it as an object, but we might want to mention the walk down the tunnel when the

TADS 3 Tour Guide

PC travels south. The simplest way to do that is with a `TravelMessage`. We do not need to define this as a separate object, it can simply be an anonymous nested object attached to the `south` property of `smallCave`:

```
smallCave : DarkRoom 'Small Cave' 'the small cave'
  "The long narrow tunnel from the south comes to an end in this cramped,
  sandy-floored cave, whose rough rocky walls press in claustrophically
  on every side. Anyone much taller than average would have to stoop here. "
  south : TravelMessage
    {
      -> secretPassage
      "You walk south for quite some way down a long tunnel. ";
    }
;
```

This time, we have used the [TravelMessage template](#) to simplify the definition here. The first template property here, `-> secretPassage`, is in fact the **destination** property of the `TravelMessage`, while the second, the double-quoted string, is its **travelDesc** property (defined on `TravelWithDesc`, from which `TravelDesc` inherits).

2.16. RoomAutoConnector

`RoomAutoConnector` is not a class that you're ever likely to use explicitly, but implicitly you'll probably use it a great deal, since it is one of the classes from which `Room` inherits. It is `RoomAutoConnector` that provides the behaviour that allows a `Room` to be used as a connector to itself. This may sound a little arcane, but in practice this is what allows us to define travel between rooms without explicitly having to define any explicit `TravelConnector` objects unless we need them for their side-effects. Since a `Room` is also a `RoomAutoConnector`, we can use as the value of another `Room`'s direction properties to implement direct travel between rooms. For example:

```
anotherCave: Room 'Another Cave'
  "There's something artificial about this cave. It's almost as if it's trying
  to be a room. The floor is suspiciously level, the walls are almost
  smooth, and there's a smart new door set into the south wall, with a
  bright electric light mounted above it. The rougher, larger central
  cave lies to the north. "
  north = mainCave
;
```

To make `anotherCave` reachable from `mainCave`, we must similarly add

```
south = anotherCave
```

to the definition of `mainCave`, which should now look like:

```
mainCave: Room 'Large Cave'
  "The flickering orange light from the blazing torch fixed to the wall
  accentuates the naturally ruddy hues of this large, irregular cave,
  which seems to be something of a major hub in the cave system. A
  large rock rests against the wall to the north, other caves lie
  through exits to east and south, while the way west is blocked by
  a huge boulder. A sturdy steel ladder leads up through a hole in the roof."
  north = rock
  south = anotherCave
  up = upLadder
;
```

2.17. Door

A basic door is easy to implement; here we'll illustrate a simple double-sided door by placing one between `anotherCave` and a room to the south that we'll call `lakeShore`. We point the `south` property of `anotherCave` to one side of the door (`lakeDoor`), the `north` property of `lakeShore` to the other side of the door (`lakeDoor2`) and make sure that one side of the door (`lakeDoor2`) points to the other side (`lakeDoor`) as its `masterObject`. `Door` inherits from `Passage` and hence from `Thing`; we use the [Thing template](#) for `lakeDoor` and the [Passage template](#) for `lakeDoor2`:

TADS 3 Tour Guide

```
anotherCave: Room 'Another Cave'
    "There's something artificial about this cave. It's almost as if it's trying
    to be a room. The floor is suspiciously level, the walls are almost
    smooth, and there's a smart new door set into the south wall, with a
    bright electric light mounted above it. The rougher, larger central
    cave lies to the north. "
    north = mainCave
    south = lakeDoor
;

+ lakeDoor : Door 'smart new door' 'smart new door';

lakeRoom: Room 'Lake Shore'
    "This is the northern shore of a giant underground lake. A door leads north. "
    north = lakeDoor2
;

+ lakeDoor2 : Door ->lakeDoor 'door' 'door';
```

Later, we'll make this more interesting by adding a special kind of lock to the door.

2.18. BasicDoor

A BasicDoor encapsulates the behaviour common to both [Door](#) and [SecretDoor](#) and their descendents, and is thus intended as an abstract class containing the common behaviour of door-like objects, rather than as a class that a game author would use directly in a game. If you wanted to a special kind of door that didn't fit either Door or SecretDoor (and their descendents) you might want to derive it from this class.

The framework provided by BasicDoor does the following:

- Provides a getFacets routine which makes both sides of a BasicDoor facets of each other (assuming one of the doors points to the other as its other side).
- Overridden makeOpen to keep both sides of a BasicDoor in sync with each other when one side is opened or closed.
- Provides routines for noting and describing a remote opening of the door (to cope with the situation where a door is opened or closed from the other side from that on which the player character is on).
- Provides handling for executing TravelVia the BasicDoor
- Boost the likelihood that this door is the object of commands like LOCK or CLOSE if this is the last door-like object the PC has traversed.

2.19. NoTravelMessage

We have described the lakeShore room as being on the northern shore of a giant underground lake. This means that it should be fairly apparent that the PC cannot proceed south. In this situation we may want to display a custom message if the player nevertheless attempts to walk out onto the lake; a NoTravelMessage will perform this role (using the [NoTravelMessage template](#)):

```
lakeRoom: Room 'Lake Shore'
    "This is the northern shore of a giant underground lake. A door leads north. "
    north = lakeDoor2
    south : NoTravelMessage { "You never learnt to walk on water. " }
    southeast asExit(south)
    southwest asExit(south)
;
```

This is very similar to a [FakeConnector](#). The only difference is a direction attached to a NoTravelMessage won't be included in a list of exits (e.g. in response to an EXITS command, or in the status line), whereas that attached to a FakeConnector will. A NoTravelMessage should therefore be used to explain why travel is not possible in a direction in which it's reasonably apparent that travel isn't possible, while a FakeConnector should be used to make travel apparently possible in a direction in which it isn't really, e.g.. to provide a "soft boundary" to the map.

2.20. ExitOnlyPassage

An ExitOnlyPassage is designed for one-way travel into a room, the other side of a passage through which you can pass but by which you cannot return. For example, imagine you have a slippery chute leading down from one cave to another below it. In the upper cave the chute might be represented by a ThroughPassage that the Player Character can enter; in the lower cave, the other end of the chute, which ejects the PC into the lower cave but can't be climbed back up could be implemented as an ExitOnlyPassage. To illustrate this we'll add four more objects: a round cave to the west of mainCave to act as the start of the chute, a long cave underneath to act as the destination of the chute, and the two halves of the chute, one in each location:

```
roundCave : DarkRoom 'Round Cave' 'the round cave'
    "This round, rocky cave has a narrow exit to the east and a strange square
    hole in the floor. "
    east = mainCave
    down = squareHole
;

+ squareHole : TravelWithMessage, ThroughPassage 'square hole/chute' 'square hole'
    "The hole is just about large enough for one person to fit through. A glint
    of something metallic can be seen just through the hole. "
    travelDesc = "You find yourself sliding down a long, slippery metal chute;
    After a short ride you are ejected into another cave. "
;

longCave : DarkRoom 'Long Cave' 'the long cave'
    "This long narrow cave runs from east to west between rough walls and
    a low ceiling. There is a large square hole in the west wall, while
    a ladder fixed to the wall at the east end runs up to a trapdoor
    set in the ceiling. Some words have been crudely scratched on the
    south wall. "
    west : NoTravelMessage { "You can't climb back up the chute, it's
    too slippery. " }
;

+ ExitOnlyPassage -> squareHole 'square hole/chute' 'square hole'
    "Through the square hole you can see the bottom end of the shiny metal
    chute, which is too slippery to climb back up. "
;
```

One other thing we need to add before this can be tested is

```
west = roundCave
```

to mainCave.

Note that we don't need to give the ExitOnlyPassage a name; we simply point it to the squareHole with the -> symbol in the [Passage template](#) to connect the two halves of the chute together; in the [Passage template](#) the -> references the masterObject property. Note also the use of a [NoTravelMessage](#) to explain why we can't climb back up the chute if we try to go west, and of the [TravelWithMessage](#) mix-in class used with the [ThroughPassage](#) to provide a description of the descent via the chute.

You can compile and run this, but you'll need to use MEGA or FIAT LUX to see what you're doing in the dark rooms.

2.21. AutoClosingDoor

So far we've provided a way of getting into the long cave, but no way of getting out. Let's suppose that the way back up is also a one-way trip, via a trapdoor in the ceiling that closes each time you go through it. This would be a good example of an AutoClosingDoor. The other (top) side of the trapdoor could be another [ExitOnlyPassage](#), since we don't want to allow the trapdoor to be opened from the upper cave (we want to force the player to use the chute we've so carefully implemented). We'll have the trapdoor open into yet another new room, a square cave to the east of mainCave:

TADS 3 Tour Guide

```
longCave : DarkRoom 'Long Cave' 'the long cave'
  "This long narrow cave runs from east to west between rough walls and
  a low ceiling. There is a large square hole in the west wall, while
  a ladder fixed to the wall at the east end runs up to a trapdoor
  set in the ceiling. Some words have been crudely scratched on the
  south wall. "
  west : NoTravelMessage { "You can't climb back up the chute, it's
    too slippery. " }
  up = longCaveLadder
;

+ ExitOnlyPassage -> squareHole 'square hole/chute' 'square hole'
  "Through the square hole you can see the bottom end of the shiny metal
  chute, which is too slippery to climb back up. "
;

+ longCaveLadder: StairwayUp 'ladder' 'ladder'
  "The ladder fixed to the east wall leads up to a trapdoor in the ceiling. "
  dobjFor(TravelVia) remapTo(TravelVia, trapdoor)
;

+ trapdoor : AutoClosingDoor 'trap trapdoor/door' 'trapdoor';

squareCave : DarkRoom 'Square Cave' 'the square cave'
  "This large square cave boasts a solitary exit to the west. "
  west = mainCave
;

+ ExitOnlyPassage -> trapdoor 'trap trapdoor/door' 'trapdoor'
  "You can hardly see the trapdoor from this side, and there is no means to
  pull it open. "
;
```

The other thing to note here is the way we've handled the ladder. We've made it a `StairwayUp` to allow it to be climbed, but it is actually the trapdoor rather than the ladder that must be traversed to reach the square cave above. There's no easy way to make the trapdoor the destination of the ladder and the other side of the trapdoor the destination of its underside. It's far easier to make traversing (i.e. climbing) the ladder equivalent to traversing (i.e. going through) the trapdoor. However such actions may be described by the player (CLIMB LADDER, CLIMB UP LADDER, ENTER TRAPDOOR, GO THROUGH TRAPDOOR) they'll end up being mapped to `TravelVia` actions internally. We can therefore simply redirect a `TravelVia` action on the ladder to a `TravelVia` action on the trapdoor, which we do using the `dobjFor` and `remapTo` macros.

Don't forget to add `east = squareCave` to the definition of `mainCave`. Then you can recompile and test the game once more.

Here the trapdoor uses the [Thing template](#) and the `ExitOnlyPassage` the [Passage template](#).

TADS 3.0.9 defines a new method on `AutoClosingDoor`, **`reportAutoClose()`**, which can be customised if we want an `AutoClosingDoor` to report its automatic closing in anything other than the default way. Suppose, for example, that when the player character goes through the trapdoor, instead of the standard "After you go through the trapdoor, it closes behind you" we want it to say, "After you emerge through the trapdoor, it slams shut behind you". You can achieve this by redefining the trapdoor thus:

```
+ trapdoor : AutoClosingDoor 'trap trapdoor/door' 'trapdoor'
  reportAutoClose = "<.p>After {you/he} emerge{s} through the trapdoor, it slams
    shut behind {it actor/him}. "
;
```

If you want an `AutoClosingDoor` to close silently (i.e. without any report at all), you can simply override `reportAutoClose()` to do nothing.

2.22. OneWayRoomConnector

Probably the most common use for a `OneWayRoomConnector` is to impose some kind of condition on traveling from one room to a second (but not the other way, or at least not symmetrically, which would call for a [RoomConnector](#)). In this situation a `OneWayRoomConnector` can be used as a nested anonymous object on one of the first room's direction properties, its `canTravelerPass` method overridden to define the conditions under which travel is possible, and its `explainTravelBarrier` method overridden to explain why travel isn't possible, if `canTravelerPass` disallows it. Travel via the `OneWayRoomConnector` is allowed if `canTravelerPass` returns true and prevented if it returns nil. Only in the latter case is `canTravelerPass` invoked to display the reason why travel has been blocked.

For example, the description of `mainCave` refers to a huge boulder blocking the exit to the west. Later, we'll implement a way of removing this obstacle by blowing it up with a stick of dynamite - so this isn't an obstacle that can readily be implemented as a [SecretDoor](#), like the rock to the north. Instead, we could add a `OneWayRoomConnector` to check whether the boulder is present, and simply disallow travel west if it is:

```
mainCave: Room 'Large Cave'
    "The flickering orange light from the blazing torch fixed to the wall
    accentuates the naturally ruddy hues of this large, irregular cave,
    which seems to be something of a major hub in the cave system. A
    large rock rests against the wall to the north, other caves lie
    through an archway to the east and an opening to the south, while
    <<boulder.moved ? 'a passage has been opened up to the west' : 'the
    way west is blocked by a huge boulder'>>. A sturdy steel ladder leads
    up through a hole in the roof. "
    north = rock
    south = anotherCave
    west : OneWayRoomConnector
        {
            ->roundCave
            canTravelerPass (traveler) { return boulder.moved; }
            explainTravelBarrier (traveler)
                { "The huge boulder is in the way. "; }
        }
    east = squareCave
    up = upLadder
;

+ boulder : Thing 'boulder' 'boulder'
    initDesc = "The huge boulder is blocking the exit to the west. "
;
```

In this case the [OneWayRoomConnector template](#) simply defines the `->` property as the destination property, so `->roundCave` means that `roundCave` is where we end up when travel is allowed via this connector. Since the only way into the `roundCave` is by going west from `mainCave`, we do not need to impose a similar check on travel the other way round; although the boulder would prevent egress from `roundCave` to `mainCave`, while the boulder is in place the player character cannot get into `roundCave` so the situation will never arise.

We have temporarily given a minimal definition of `boulder` simply as a `Thing` so that it can readily be removed to allow access to the `roundCave`. We shall change this when we come to implement the means of blowing it up. Note the use of `initDesc` to give an appropriate description of the boulder before it is moved, and the alteration to the room description so that it changes when the boulder is removed.

2.23. PathPassage

A `PathPassage` is intended for use as an outdoor passage such as a road or path that is not enclosed. It is basically the same as a [ThroughPassage](#) apart from the way that travel via it is described (when an actor other than the PC goes along it). Another, and perhaps more interesting, feature of the `PathPassage` is that the English Language extensions to the library understand the command `TAKE PATH` in the sense of 'walk along the path' instead of 'pick up the path'. We can try this out by adding a short path along the side of the lake:

TADS 3 Tour Guide

```
lakeRoom: Room 'Lake Shore' 'the lake shore'
    "This is the northern shore of a giant underground lake. A door leads north,
    and a path runs a short way east. "
    north = lakeDoor2
    south : NoTravelMessage { "You never learnt to walk on water. " }
    southeast asExit(south)
    southwest asExit(south)
    east = lakePath
;

+ lakeDoor2 : Door ->lakeDoor 'door' 'door';

+ lakePath : PathPassage 'short eastward rocky lakeside path' 'short lakeside path'
    "The rocky path runs short way along the side of the lake. "
;

pathEnd : OutdoorRoom 'End of Lakeside Path' 'the end of the path'
    "The path from the west comes to an end just here, on the northern
    shore of the great underground lake. "
    west = lakePath2
    south : NoTravelMessage { "The lake is in the way. " }
;

+ lakePath2 : PathPassage ->lakePath 'westward lakeside path' 'westward path'
    "The path leads off along the shore of the lake to the west. "
;
```

If you compile and run the game, you should be able to type SOUTH, DOWN, SOUTH, SOUTH (as four separate commands) to arrive at the lakeside. From there you can type TAKE PATH to travel to pathEnd. Typing TAKE PATH a second time will return you to lakeRoom.

Note that PathPassage uses the same [templates as Passage](#).

The PathPassage class provides a convenient opportunity to introduce another library feature, albeit one that's only tangentially related. In English, the expression TAKE PATH can mean FOLLOW PATH (i.e. go down the path, travel via the path), and hence the English language part of the library defines:

```
modify PathPassage
/* treat "take path" the same as "enter path" or "go through path" */
dobjFor(Take) remapTo(TravelVia, self)
;
```

The problem with this is that while TAKE PATH might mean FOLLOW PATH, GET PATH or PICK UP PATH do not, and yet all three forms of the command will match TakeAction. It's true that the commands GET PATH or PICK PATH UP don't make much sense, but it may still be a bit puzzling to players if they're treated as instructions to wander down the path. What we'd really like here is a means of distinguishing between TAKE PATH on the one hand and GET PATH or PICK PATH UP on the other. In other words, it would be helpful to know what phrasing the player used in actually issuing the command in this particular case, without going to the trouble of having to create a separate GetAction which would be treated as equivalent to TakeAction in 98% of cases.

As from TADS 3.0.10 the library provides a solution to this in the form of an action method **getEnteredVerbPhrase()**. We can call this on gAction to return a string containing the exact *verb* phrasing, but with the direct and (if present) indirect objects replaced with the placeholder tokens '(dobj)' and '(obj)'. So, for an example, if the player had typed PUT BIG RED BALL IN THE SMALL PLASTIC BUCKET, gAction.getEnteredVerbPhrase would return the string 'put (dobj) in (obj)', which shows us the structure of the command used without worrying about the wording used to describe the objects involved, and without worrying about whether the player typed the command in lower case or upper case or a mixture of the two, since getEnteredVerbPhrase returns a string converted entirely to lower case (if we actually wanted the full original phrasing of the command we could use **gAction.getOrigText()** instead).

We could use getEnteredVerbPhrase to distinguish between TAKE PATH and GET PATH or PICK UP PATH:

```
modify PathPassage
dobjFor(Take)
{
    remap()
    {
        if(gAction.getEnteredVerbPhrase == 'take (dobj)')
            return [TravelViaAction, self];
        else

```

TADS 3 Tour Guide

```
        return nil;
    }
}
;
```

Note that in this case we couldn't use the `maybeRemapTo` macro, since if the condition failed we'd get the inherited handling, which remaps `Take` to `TravelVia` unconditionally, thus defeating the object of the modification. In other cases, however, we could use `maybeRemapTo`. For example, we might want `TAKE PILL` to be remapped to `EAT PILL`, but not `GET PILL` or `PICK UP PILL`:

```
pill: Edible 'little green pill*pills' 'little green pill'
  dobjFor(Take) maybeRemapTo(gAction.getEnteredVerbPhrase == 'take (dobj)', Eat, self)

  dobjFor(Eat)
  {
    action()
    {
      "It tastes absolutely vile -- which presumably means that it's meant to be good for you.
      You force yourself to swallow it nonetheless. ";
      inherited;
    }
  }
;
```

In this case the remap to `EAT PILL` will work whether the player types `TAKE PILL` or `TAKE THE GREEN PILL` or `TAKE LITTLE GREEN PILL` or any other such combination.

2.24. Shipboard

`Shipboard` is a mix-in class that can be added to other room classes to indicate that shipboard directions (port, starboard, fore and aft) are meaningful in such locations. Clearly, the principal use of this class will be when constructing locations aboard a ship.

To illustrate the use of this class, we first need a ship. Fortunately, we already have a lake we can float it on, so we can begin by defining it thus:

```
ship : Enterable ->portDeck 'large wooden sailing ship' 'ship' @lakeRoom
  "It's a large wooden sailing ship, close enough to the shore to board. "
  specialDesc = "A large wooden ship floats on the lake, just by the shore. "
  dobjFor(Board) asDobjFor(Enter)
  getFacets { return [leaveShip]; }
;
```

There are a number of points to note about this short definition. The first is the use of the `@` notation as an alternative means of specifying the ship's initial location. Although the ship is notionally on the lake, we in fact place it in `lakeRoom` since we want it to be visible and enterable from there (before it moves). However, since we always want the ship to be *described* as floating on the lake, we add a `specialDesc` to that effect; this is how the ship will then always be described when listed in room descriptions. Although the ship will not always remain in *this* location, it will always be in some location close to the shore, and our description is sufficiently general to cover that. Next, a player is as likely to type `BOARD SHIP` as `ENTER SHIP` in order to enter the vessel, so we add `dobjFor(Board)` as `dobjFor(Enter)` to make `BOARD` equivalent to `ENTER` here. Finally, we'll assume that on boarding the ship we arrive on the port deck, so we use the `->` notation of the `Enterable` template to indicate that `portDeck` is the location this `Enterable` takes us to. We'll explain the mysterious **getFacets** in just a minute.

We next need to define the `portDeck` location. Since there'll be several other deck locations, all of which will use the `Shipboard` mix-in class, we can save ourselves a bit of typing if we first define a custom `Deck` class:

```
class Deck : Shipboard, OutdoorRoom;
```

We can then define the `portDeck` thus:

```
portDeck : Deck 'Port Deck' 'the main deck'
  "This part of the main deck is on the port side of the ship, close to the shore. The
  deck continues to fore, aft and starboard, and a tall mast towers up from
```

TADS 3 Tour Guide

```
the middle of the main deck. "
fore = foreDeck
aft = quarterDeck
starboard = starboardDeck
out = (ship.location)
up = mast
;
```

We'll be defining the destinations referred to shortly; for now the only one to note is that attached to the `out` property. This is set to `(ship.location)` so that whenever we type OUT from `portDeck` we'll end up in whatever location the ship object is in; this provides an easy way of moving the entire ship. But of course, just as players may type BOARD SHIP to enter the ship, they may also want to type LEAVE SHIP or GET OUT OF SHIP to disembark. The way to handle this is to provide an `Exitable` object for SHIP to refer to in these circumstances:

```
+ leaveShip : Exitable ->(ship.location) 'ship' 'ship'
  "It's a large wooden sailing vessel, which stretches fore, aft and to starboard of
  the port deck. "
  getFacets { return [ship]; }
;
```

Note that `(ship.location)` needs to be enclosed in parentheses when using the template `->connector` syntax here, since it is an expression. Otherwise, the `Exitable` behaves pretty much the same way as the [Enterable](#) we encountered before (except that it handles EXIT so-and-so instead of ENTER so-and-so). The main point to note here is the use of the `getFacets` method. The point of this is that although they are separate programming objects, both `ship` and `leaveShip` refer to the same physical object. In this case the two programming objects could be regarded as two different facets of the same ship seen from the shore or from its port deck. The `getFacets` method, which returns a list of the other facets of an object, is the means by which we can specify this relation to the parser. The practical effect of this is that the player can type BOARD SHIP followed by LEAVE IT, and the parser will be able to work out that IT should now refer to the `leaveShip` object. Without the use of `getFacets` the LEAVE IT command would fail, since the original ship would no longer be in scope to be the object of the LEAVE command, and would not handle the command properly even if it were. Note that for doors and other passage-like objects that the library recognizes as double-sided entities this `getFacets` mechanism is automatically set up by the library, so it is only in less standard contexts such as the present one that game authors need to worry about it.

With these complications out of the way, the definition of the starboard part of the deck is fairly straightforward:

```
starboardDeck : Deck 'Starboard Deck' 'the main deck'
  "From the starboard side of the ship there's a clear view over the lake as far as
  the eye can see to starboard. The deck continues forward, aft and to port, and a
  tall mast rises up from the centre of the main deck. "
  port = portDeck
  fore = foreDeck
  aft = quarterDeck
  up = mast
;
```

The `foreDeck` and `quarterDeck` would be even more straightforward but for one complication. The way we have defined our ship, the main deck straddles its central portion and is divided into a port side and a starboard side. Going aft from either side takes us to the `quarterDeck`, which going forward from either side takes us to the `foreDeck`. So where should we end up if we come aft from the `foreDeck` or forward from the `quarterDeck`? Clearly we should arrive somewhere on the main deck, but should it be on the port or the starboard side? It could be either but there seems no clear reason why it should be on or the other. One way to handle this is for travel aft from the `foreDeck` or forward from the `quarterDeck` to bring the PC to either location, chosen at random on each location (which will also give the player something else to figure out!). The neatest way to implement that is by using a `OneWayRoomConnector` that produces this result:

```
mainDeck : OneWayRoomConnector
  destination = (rand(portDeck, starboardDeck))
;
```

The definition of the other two deck sections then becomes straightforward:

```
foreDeck : Deck 'Fore Deck' 'the fore deck'
  "The foredeck is at the front of the ship, overlooking the bows. Most of the
  ship is aft from here. "
  aft = mainDeck
;
```

TADS 3 Tour Guide

```
quarterDeck : Deck 'Quarterdeck' 'the quarterdeck'
    "The quarterdeck is a raised portion of the deck near the stern of the ship, and separated
    from the deck further foreward by a wooden rail on which is mounted a panel. A flight
    of steps leads down below. "
    fore = mainDeck
    down = deckSteps
;

+ deckSteps : StairwayDown 'flight steps' 'steps'
    "The steps lead down into a cabin below. "
    isPlural = true
;
```

That said, when the player goes fore from the quarterDeck, it will normally be with the intention of leaving the ship, via the port deck. The random selection of destinations in this case will quickly become an annoyance, so it is probably better to define:

```
quarterDeck : Deck 'Quarterdeck' 'the quarterdeck'
    "The quarterdeck is a raised portion of the deck near the stern of the ship, and separated
    from the deck further foreward by a wooden rail on which is mounted a panel. A flight
    of steps leads down below. "
    fore = portDeck
    port = portDeck
    starboard = starboardDeck
    down = deckSteps
;
```

Note: Shipboard and ShipboardRoom: prior to TADS 3.0.8 the class now called Shipboard was called ShipboardRoom. The name was changed because Shipboard is a mix-in class designed to be used with Rooms, but not actually a type of Room, so it seemed illogical to include Room in its name.

TADS 3.0.8 still defines a ShipboardRoom class (for convenience), but it now means something slightly different, being defined simply as a combination of Shipboard and Room:

```
class ShipboardRoom : Shipboard, Room
;
```

2.25. FloorlessRoom

A FloorlessRoom, as its name suggests, is a location that lacks a floor, such as the top of a vertical shaft, or a tree. The top of a mast, which is the sort of thing one would expect to find aboard ship, is another good example. Apart from lacking a floor (something we'll discuss in more detail when we come to talk about roomParts) a FloorlessRoom has the property that something dropped there does not remain there but drops out of sight either into oblivion, or into other specified location (such as the bottom of the vertical shaft, tree or mast).

Before we can define the top of the mast we need to define its bottom. We'll assume the mast is located in the centre of main deck, i.e. between portDeck and starboardDeck. It thus exists in both locations, which makes it an ideal example of a [MultiLoc](#):

```
mast : MultiLoc, StairwayUp 'tall thick wooden mast' 'tall mast'
    "The thick wooden mast towers up at least a hundred feet. "
    locationList = [portDeck, starboardDeck]
;
```

We also make the mast a [StairwayUp](#), since although it does not look much like a flight of stairs, it is something we can climb to reach another location, and so it behaves like a StairwayUp. Note that the MultiLoc mix-in class must be specified before the Thing-derived class (in this case StairwayUp) in the list of superclasses, and that its locationList property contains a list of locations where the mast can be found.

We can now define the top of the mast as our FloorlessRoom example. The one thing we need to consider is how we want to specify its bottomRoom property (the place where objects dropped here will end up). One would expect something dropped from the top of the mast to fall to the deck below, but should it land in portDeck or starboardDeck? Likewise, where should the PC fetch up when he or she descends the mast? This is precisely the same dilemma we had when deciding how to proceed aft from foreDeck, so we can use precisely the same solution:

TADS 3 Tour Guide

```
topOfMast : FloorlessRoom 'Top of Mast' 'the top of the mast'
    "From the top of the mast you can see miles out across the lake to starboard
    and the shore over to port. The deck below looks a sickenly long way down. "
    down = mainDeck
    bottomRoom = (mainDeck.destination)
;
```

The result of this is that something dropped from the top of the mast has an equal chance of fetching up in portDeck or starboardDeck (in a simpler situation we could simply have specified a single room as the value of the bottomRoom property). In a moment or two you can test this out by picking up the boulder on the way to the ship and dropping it from the top of the mast. But first we have one more task to complete, and that is to provide a mast object for the PC to climb down from at the top of the mast. This should clearly be a [StairwayDown](#), but the problem is that its masterObject will be the MultiLoc mast object, so it won't be able to handle climbing down properly - indeed, unless we do something about it we'll get a runtime error before we even try. The solution is to remap the StairwayDown's TravelVia handling to the mainDeck connector:

```
+ StairwayDown ->mast 'mast' 'mast'
    "Right now you're clinging to it for dear life. "
    dobjFor(TravelVia) remapTo(TravelVia, mainDeck)
;
```

You should now be able to compile and run the game to test that everything is working correctly. When moving about the deck you can abbreviate the PORT, STARBOARD, FORE and AFT commands to P, SB, F and A respectively. Just don't try going down the steps from the quarterdeck yet.

2.26. Floorless

Floorless is a mix-in class which adds Floorless behaviour to any Room class; that is it takes away the floor from the list of [room parts](#), and provides the handling for a dropped object to end up in another location.

Since the top of the mast is not exactly an (indoor) Room, in the sense of having four walls and a ceiling, it would be better defined using a mix of Floorless and a more appropriate class:

```
topOfMast : Floorless, Deck 'Top of Mast' 'the top of the mast'
    "From the top of the mast you can see miles out across the lake to starboard
    and the shore over to port. The deck below looks a sickenly long way down. "
    down = mainDeck
    bottomRoom = (mainDeck.destination)
;
```

Although the randomizing maindeck connector works fine, as you'll have seen if you've experiment with it, it is actually not a very good idea in practice. Not only will the random connector be potentially confusing to players, when we come to define an NPC who will follow the PC around, it can result in accidentally losing her (for example if the PC goes south from the foredeck and the NPC tries to follow him, she may end up on the port deck and he on the starboard, which is simply wrong if she's meant to be following him). Thus, once you've experimented with this random connector (assuming you want to), I suggest you remove its random element and change it to:

```
mainDeck : OneWayRoomConnector
    destination = portDeck
;
```

2.27. HiddenDoor

A HiddenDoor is a variation on SecretDoor, the difference being that while a [SecretDoor](#) is a visible object (like the rock we used before) that is not apparently a door, a HiddenDoor isn't even visible until it's been opened. For our example we'll create a section of the foreward bulkhead of the cabin that slides open at the press of a button. We'll go about concealing the button in a later section.

First, however, we need to create the cabin:

TADS 3 Tour Guide

```
class Cabin : ShipboardRoom, Room;

greatCabin : Cabin 'Great Cabin' 'the great cabin'
    "The great cabin occupies the entire width of the ship at the stern. The stern
    windows aft look out over the water, while there is a solid wooden bulkhead
    foreward. The main piece of furniture is a sturdy wooden desk, while a flight of
    steps leads up to the deck above. "
    up = cabinSteps
    fore = bulkheadDoor
;

+ cabinSteps : StairwayUp -> deckSteps 'flight steps' 'steps'
    "The steps lead up to the deck above. "
    isPlural = true
;
```

There is nothing new in this, apart from the creation of our custom Cabin class (which, along with the Deck class, we'll shortly be customizing a little further). We can now define the HiddenDoor:

```
+ bulkheadDoor : HiddenDoor 'bulkhead door/doorway/opening' 'bulkhead door'
    "The central section of the foreward bulkhead has slid open, revealing a
    doorway through the bulkhead. "
    destination = crewQuarters
;
```

We next need to provide a mechanism for opening it, which we'll make a button that, for now, is simply a Fixture in the cabin:

```
+ Button, Fixture 'small brown button' 'small brown button'
    "The small brown button is fixed to the underside of the desk. "
    dobjFor(Push)
    {
        action()
        {
            "There's a sharp <i>click</i>, and a section of the foreward bulkhead slides
            <<bulkheadDoor.isOpen ? 'closed' : 'open'>>. ";
            bulkheadDoor.makeOpen(!bulkheadDoor.isOpen);
        }
    }
;
```

The description of the button shows where we'll end up putting it, but that will come later. Finally, we need to define another couple of rooms where we fetch up when we go through the HiddenDoor:

```
class DarkCabin : Cabin
    brightness = 0
;

crewQuarters : DarkCabin 'Crew Quarters' 'the crew quarters'
    "The crew quarters seem largely deserted. There's an exit back aft and a
    ladder leading down into the hold. "
    down = holdLadderDown
    aft = greatCabin
;

+ holdLadderDown : StairwayDown 'ladder' 'ladder';

hold : DarkCabin 'Hold'
    "The hold seems vast and cavernous, and is largely empty. A ladder leads
    up through an open hatchway above. "
    up = holdLadderUp
;

+ holdLadderUp : StairwayUp -> holdLadderDown 'ladder' 'ladder';
```

We could have defined DarkCabin as ShipboardRoom, DarkRoom; but by making it inherit from Cabin we ensure that it inherits any further customizations we add to the Cabin class.

2.28. EntryPortal

An EntryPortal is just like an [Enterable](#), except that you can go through it as well as enter it. It can be used, for example, for an archway that is plainly the entrance to another destination:

```
mainCave: Room 'Large Cave'
    "This is the main cave. A large rock rests against the north wall and
    there is another cave to the south and an archway to the east,
    <<boulder.moved ? 'and a passage
    has been opened up to the west' : 'but the way west is blocked by
    a huge boulder'>>. A blazing torch is fixed to the wall, next to a sturdy
    steel ladder leading upwards. "
    north = rock
    south = anotherCave
    west : OneWayRoomConnector
        {
            ->roundCave
            canTravelerPass (traveler) { return boulder.moved; }
            explainTravelBarrier (traveler)
                { "The huge boulder is in the way. "; }
        }
    east = squareCave
    up = upLadder
;

+ EntryPortal ->squareCave 'arch/archway' 'archway'
    "It's a large archway, leading to another cave beyond. "
;
```

The property pointed to by -> in the template is actually the connector traversed, not the destination reached, when the EntryPortal is entered, although when, as here, the connector is a Room this has the same effect (see this discussion of the distinction in connection with the [Enterable](#) class, from which EntryPortal inherits). Entry portal inherits from Enterable and hence inherits the [Enterable template](#).

2.29. ExitPortal

An ExitPortal is just like an [Exitable](#), except that you can go through it as well as exit it. For example:

```
squareCave : DarkRoom 'Square Cave' 'the square cave'
    "This capacious cave is unnaturally square, suggesting that it has been
    artificially hewn out of the rock, an impression further enhanced by
    the carefully-constructed ashlar archway to the west. "
    west = mainCave
    out asExit(west)
;

+ ExitPortal -> mainCave 'ashlar arch/archway' 'archway'
    "The archway is beautifully constructed from dressed stones.
;
```

Note that ExitPortal is *not* a travel connector; -> mainCave makes mainCave its connector property, not its masterObject property. For the distinction, see further on the [Enterable](#) class. The template used here is the [Exitable template](#).

2.30. TravelBarrier

In all the examples we have used so far, when we have wanted to prevent travel via a TravelConnector, we have overridden its `canTravelerPass` method to determine whether travel is permitted, and its `explainTravelBarrier` method to explain why travel is forbidden (if it is forbidden). Normally this is the simplest and most convenient way to do it - but there is another way, and that is to use TravelBarrier object.

A TravelBarrier is simply an object that defines `canTravelerPass` and `explainTravelBarrier` methods. A single TravelBarrier, or a list of TravelBarriers, can be attached to a TravelConnector via its `travelBarrier` property. This can be useful in a number of cases.

The first case is when a specialized type of TravelBarrier, such as the [PushTravelBarrier](#), is required.

The second case is where you want to enforce the same barrier conditions on a number of different TravelConnectors. Rather than write the same `canTravelerPass` and `explainTravelBarrier` methods on two or more TravelConnectors, you can define them once on a TravelBarrier object then attach the object to each of the TravelConnectors to which it applies. For example, suppose you want to prevent the player traveling either north or east from a particular location without the lamp, you could define:

```
lampBarrier : TravelBarrier
  canTravelerPass(traveler) {return lamp.isIn(traveler); }
  explainTravelBarrier(traveler) { "You forgot the lamp! "; }
;
```

Then, on the relevant location you could define:

```
north : OneWayRoomConnector { -> darkPassage travelBarrier = lampBarrier }
east : OneWayRoomConnector { -> darkCorridor travelBarrier = lampBarrier }
south = lampRoom
```

The third case is where you want to perform a number of separate checks, each of which would result in a different failure message. Rather than write a long switch statement or series of if statements in the `explainTravelBarrier` method of the TravelConnector, you could define a number of TravelBarrier objects that pair the condition with the message. For example, supposing that at another point in your journey, you want not only to enforce the condition that the player has the lamp, as above, but also that he's not wearing the stolen jacket. You might then define another TravelBarrier object:

```
jacketBarrier : TravelBarrier
  canTravelerPass(traveler) {return !jacket.isWornBy(traveler); }
  explainTravelBarrier(traveler) { "You'll stand out a mile wearing Lord Ponsonby's jacket in there! "; }
;
```

Then you can attach both TravelBarriers to the same connector:

```
in : OneWayRoomConnector { -> pompousClubLobby
  travelBarrier = [lampBarrier, jacketBarrier ]
}
```

What happens is that among the checks carried out in the `checkObjTravelVia` method of a TravelConnector is a call to `checkTravelBarriers`; this first checks the `canTravelerPass` method of the TravelConnector itself, then works through the list of TravelBarriers (if any) in the `travelBarrier` property, calling each of their `canTravelerPass` methods in turn. If any of these `canTravelerPass` methods returns nil, the travel is aborted and the corresponding `explainTravelBarrier` method is called.

2.31. AskConnector

The normal IF convention assumes that there is only one exit in any given compass direction. But what happens if you want to model a situation where there are two, or three, or half a dozen, such as a north wall in which there are several doors? You could, of course, simply attach a FakeConnector to the north property of such a location and have it display a message telling the player to select a door to go through instead, but a better solution would be to use an AskConnector. This is an "ask which" travel connector. Rather than just traversing the connector, we ask for a direct

TADS 3 Tour Guide

object for a specified travel verb; if the player supplies the missing indirect object (or if the parser can automatically choose a default), we'll perform the travel verb using that direct object.

AskConnector defines the following properties:

- **promptMessage** - An extra prompt message to show before the normal parser prompt for a missing or ambiguous object. We'll show this just before the normal parser message, if it's specified. If you want to customize the messages more completely, you can override askDisambig() or askMissingObject(). The parser will invoke these to generate the prompt, so you can customize the entire messages by overriding these.
- **travelAction** - The specific travel action to attempt. This must be a TAction - an action that takes a direct object (and only a direct object). The default is TravelVia, but this should usually be customized in each instance to the type of travel appropriate for the possible connectors.
- **travelObjs** - The list of possible direct objects for the travel action. If this is nil, we'll simply treat the direct object of the travelAction as completely missing, forcing the parser to either find a default or ask the player for the missing object. If the travel is limited to a specific set of objects (for example, if there are two doors leading north, and we want to ask which one to use), this should be set to the list of possible objects; the parser will then use the ambiguous noun phrase rules instead of the missing noun phrase rules to ask the player for more information.
- **travelObjsPhrase** - The phrase to use in the disambiguation question to ask which of the travelObjs entries is to be used. The language-specific module provides a suitable default, but this should usually be overridden if travelObjs is overridden.

Here's an example of an AskConnector when there are two doors in the south wall.

```
stoneLanding : Room 'Landing' 'the landing'
  "A pair of doors lead south from this narrow landing, from which
  a narrow flight of stone steps lead down to the north. "
  down = slStairsDown
  north asExit(down)
  south : AskConnector
  {
    promptMessage = "There are two doors you could go through to the south . "
    travelAction = GoThroughAction
    travelObjs = [leftDoor, rightDoor]
    travelObjsPhrase = 'of them'
  }
;

+ leftDoor : Door 'left hand door*doors' 'left hand door'
;

+ rightDoor : Door 'right hand door*doors' 'right hand door'
;
```

Now, if you arrive at this destination and type the command SOUTH the parser will respond with "There are two doors you could go through to the south. Which of them do you mean, the right hand door, or the left hand door?" Of course, right now there's no way of reaching this location; we'll eventually provide it when we come to look at [Consultable](#).

2.32. TravelConnector

TravelConnector is the base class from which all the connectors we have been looking at ultimately derive. You will probably not define any objects of the TravelConnector class (as opposed to one of its subclasses) in your games, since a raw TravelConnector doesn't actually lead anywhere. It's possible that you might define your own subclass of TravelConnector for some particular purpose, or you could use a TravelConnector object in your game and override, say, its actionDobjTravelVia method or its getDestinationMethod to produce the result you want (though in most cases you'll probably want to use one of its subclasses rather than reinventing a wheel that's already in the library).

The main importance of TravelConnector, however, is that it defines a large number of the properties and methods used on all its subclasses. These are listed below for the sake of reference, using descriptions taken from the comments in the library code:

TADS 3 Tour Guide

Properties:

connectorStagingLocation: The "staging location" for travel through this connector. By default, if we have a location, that's our staging location; if we don't have a location (in which case we probably have an outermost room), we don't have a staging location.

isCircularPassage: Is this a "circular" passage? A circular passage is one that explicitly connects back to its origin, so that traveling through the connector leaves us where we started. When a passage is marked as circular, we'll describe travel through the passage exactly as though we had actually gone somewhere. By default, if traveling through a passage leaves us where we started, we assume that nothing happened, so we don't describe any travel. Circular passages don't often occur in ordinary settings; these are mostly useful in disorienting environments, such as twisty cave networks, where a passage between locations can change direction and even loop back on itself.

isConnectorListed: Is this connector listed? This indicates whether or not the exit is allowed to be displayed in lists of exits, such as in the status line or in "you can't go that way" messages. By default, all exits are allowed to appear in listings.

Note that this indicates if listing is ALLOWED - it doesn't guarantee that listing actually occurs. A connector can be listed only if this is true, AND the point-of-view actor for the listing can perceive the exit (which means that `isConnectorApparent` must return true, and there must be sufficient light to see the exit).

travelBarrier: Barrier or barriers to travel. This property can be set to a single [TravelBarrier](#) object or to a list of `TravelBarrier` objects. `checkTravelBarriers()` checks each barrier specified here.

travelMemory: Our "travel memory" table. If this contains a non-nil lookup table object, we'll store a record of each successful traversal of a travel connector here - we'll record the destination keyed by the combination of actor, origin, and connector, so that we can later check to see if the actor has any memory of where a given connector goes from a given origin. * We keep this information by default, which is why we statically create the table here. Keeping this information does involve some overhead, so some authors might want to get rid of this table (by setting the property to nil) if the game doesn't make any use of the information. Note that this table is stored just once, in the `TravelConnector` class itself - there's not a separate table per connector.

Methods:

actorTravelPreCond (actor): Get the travel preconditions that this connector requires for travel by the given actor. In most cases, this won't depend on the actor, but it's provided as a parameter anyway; in most cases, this will just apply the conditions that are relevant to actors as travelers.

By default, we require actors to be "travel ready" before traversing a connector. The exact meaning of "travel ready" is provided by the actor's immediate location, but it usually simply means that the actor is standing. This ensures that the actor isn't sitting in a chair or lying down or something like that. Some connectors might not require this, so this routine can be overridden per connector.

Note that this will only be called when an actor is the traveler. When a vehicle or other kind of traveler is doing the travel, this will not be invoked.

canTravelerPass (traveler): Check to see if the `Traveler` object is allowed to travel through this connector. Returns true if travel is allowed, nil if not.

This is called from `checkTravelBarriers()` to check any conditions coded directly into the `TravelConnector`. By default, we simply return true; subclasses can override this to apply special conditions.

If an override wants to disallow travel, it should return nil here, and then provide an override for `explainTravelBarrier()` to provide a descriptive message explaining why the travel isn't allowed.

Conditions here serve essentially the same purpose as barrier conditions. The purpose of providing this additional place for the same type of conditions is simply to improve the convenience of defining travel conditions for cases where barriers are unnecessary. The main benefit of using a barrier is that the same barrier object can be re-used with multiple connectors, so if the same set of travel conditions apply to several different connectors, barriers allow the logic to be defined once in a single barrier object and then re-used easily in each place it's needed. However, when a particular condition is needed in only one place, creating a barrier to represent the condition is a bit verbose; in such cases, the condition can be placed in this method more conveniently.

checkTravelBarriers (dest): Check barriers. The `TravelVia` `check()` routine must call this to enforce barriers.

connectorBack (traveler, dest): Find a connector in the destination location that connects back as the source of travel from the given connector when traversed from the source location. Returns nil if there is no such connector. This must be called while the traveler is still in the source location; we'll attempt to find the connector back to the traveler's current location.

The purpose of this routine is to identify the connector by which the traveler arrives in the new location. This can be used, for example, to generate a connector-specific message describing the traveler's emergence from the connector (so we can say one thing if the traveler arrives via a door, and another if the traveler arrives by climbing up a ladder).

TADS 3 Tour Guide

By default, we'll try to find a travel link in the destination that links us back to this same connector, in which case we'll return 'self' as the connector from which the traveler emerges in the new location. Failing that, we'll look for a travel link whose apparent source is the origin location. This should be overridden for any connector with an explicit complementary connector. For example, it is common to implement a door using a pair of objects, one representing each side of the door; in such cases, each door object would simply return its twin here. Note that a complementary connector doesn't actually have to go anywhere, since it's still useful to have a connector back simply for describing travelers arriving on the connector.

This *must* be overridden when the destination location doesn't have a simple connector whose apparent source is this connector, because in such cases we won't be able to find the reverse connector with our direction search.

connectorGetConnectorTo (origin, traveler, dest): Get the travel connector leading to the given destination from the given origin and for the given travel. Return nil if we don't know a connector leading there.

By default, we simply return 'self' if our destination is the given destination, or nil if not.

Some subclasses might encapsulate one or more "secondary" connectors - that is, the main connector might choose among multiple other connectors. In these cases, the secondary connectors typically won't be linked to directions on their own, so the room can't see them directly - it can only find them through us, since we're effectively a wrapper for the secondary connectors. In these cases, we won't have any single destination ourselves, so `getDestination()` will have to return nil. But we *can* work backwards: given a destination, we can find the secondary connector that points to that destination. That's what this routine is for.

connectorTravelPreCond (): Get any connector-specific pre-conditions for travel via this connector.

createUnlistedProxy (): Get an unlisted proxy for this connector. This is normally called from the `asExit()` macro to set up one room exit direction as an unlisted synonym for another.

darkTravel (actor, dest): Handle travel in the dark. Specifically, this is called when an actor attempts travel from one dark location to another dark location. (We don't invoke this in any other case: light-to-light, light-to-dark, and dark-to-light travel are all allowed without any special checks.)

By default, we will prohibit dark-to-dark travel by calling the location's `darkTravel` handler. Individual connectors can override this to allow such travel or apply different handling.

describeArrival (traveler, origin, dest): Describe an actor's arrival through the connector from the given origin into the given destination. This description is from the point of view of another actor in the destination.

Note that this is called on the connector that reverses the travel, NOT on the connector the actor is actually traversing - that is, 'self' is the backwards connector, leading from the destination back to the origin location. So, if we have two sides to a door, and the actor traverses the first side, this will be called on the second side - the one that links the destination back to the origin.

describeDeparture (traveler, origin, dest): Describe an actor's departure through the connector from the given origin to the given destination. This description is from the point of view of another actor in the origin location.

describeLocalArrival (traveler, origin, dest): Describe a "local arrival" via this connector. This is called when the traveler moves around entirely within the field of view of the player character - that is, the traveler's origin is visible to the player character when we arrive in our destination. We'll describe the travel not in terms of arriving, since the traveler was already here to start with, but rather as entering the destination.

dobjFor (TravelVia): Action handler for the internal "TravelVia" action. This is not a real action, but is instead a pseudo-action that we implement generically for travel via the connector. Subclasses that want to handle real actions by traveling via the connector can use `remapTo(TravelVia)` to implement the real action handlers. Note that `remapTo` should be used (rather than, say, `asDobjFor`), since this will ensure that every type of travel through the connector actually looks like a `TravelVia` action, which is useful for intercepting travel actions generically in other code.

explainTravelBarrier (traveler): Explain why `canTravelerPass()` returned nil. This is called to display an explanation of why travel is not allowed by `self.canTravelerPass()`.

Since the default `canTravelerPass()` always allows travel, the default implementation of this method does nothing. Whenever `canTravelerPass()` is overridden to return nil, this should also be overridden to provide an appropriate explanation.

fixedSource (dest, traveler) Get the "fixed" source for travelers emerging from this connector, if possible. This can return nil if the connector does not have a fixed relationship with another connector.

The purpose of this routine is to find complementary connectors for simple static map connections. This is especially useful for direct room-to-room connections.

When a connector relationship other than a simple static mapping exists, the connectors must generally override `connectorBack()`, in which case this routine will not be needed (at least, this routine won't be needed as long as the

TADS 3 Tour Guide

overridden connectorBack() doesn't call it). Whenever it is not clear how to implement this routine, don't - implement connectorBack() instead.

getApparentDestination (origin, actor): Get the apparent destination of travel by the actor to the given origin. This returns the location to which the connector travels, AS FAR AS THE ACTOR KNOWS. If the actor does not know and cannot tell where the connector leads, this should return nil.

Note that this method does NOT necessarily return the actual destination, because we obviously can't know the destination for certain until we traverse the connection. Rather, the point of this routine is to return as much information as the actor is supposed to have. This can be used for purposes like auto-mapping, where we'd want to show what the player character knows of the map, and NPC goal-seeking, where an NPC tries to figure out how to get from one point to another based on the NPC's knowledge of the map. In these sorts of applications, it's important to use only knowledge that the actor is supposed to have within the parameters of the simulation.

Callers should always test isConnectorApparent() before calling this routine. This routine does not check to ensure that the connector is apparent, so it could return misleading information if used independently of isConnectorApparent(); for example, if the connector *formerly* worked but has now disappeared, and the actor has a memory of the former destination, we'll return the remembered destination.

The actor can know the destination by a number of means:

1. The location is familiar to the character. For example, if the setting is the character's own house, the character would obviously know the house well, so would know where you'd end up going east from the living room or south from the kitchen. We use the origin method actorKnowsDestination() to determine this.
2. The destination is readily visible from the origin location, or is clearly marked. For example, in an outdoor setting, it might be clear that going east from the field takes you to the hilltop. In an indoor setting, an open passage might make it clear that going east from the living room takes you to the dining room. We use the origin method actorKnowsDestination() to determine this.
3. The actor has been through the connector already in the course of the game, and so remembers the connection by virtue of recent experience. If our travelMemory class property is set to a non-nil lookup table object, then we'll automatically use the lookup table to remember the destination each time an actor travels via a connector, and use this information by default to provide apparent destination information.

getDestination (origin, traveler): Get our destination, given the traveler and the origin location.

This method is required to return the current destination for the travel. If the connector doesn't go anywhere, this should return nil. The results of this method must be stable for the extent of a turn, up until the time travel actually occurs; in other words, it must be possible to call this routine simply for information purposes, to determine where the travel will end up.

This method should not trigger any side effects, since it's necessary to be able to call this method more than once in the course of a given travel command. If it's necessary to trigger side effects when the connector is actually traversed, apply the side effects in noteTraversal().

For auto-mapping and the like, note that getApparentDestination() is a better choice, since this method has internal information that might not be apparent to the characters in the game and thus shouldn't be revealed through something like an auto-map. This method is intended for internal use in the course of processing a travel action, since it knows the true destination of the travel.

Note that on the TravelConnector class this method simply returns nil, which is why a raw TravelConnector won't get you anywhere. This method is overridden on subclasses to do something more useful.

isConnectorApparent (origin, actor): Determine if the travel connection is apparent - as a travel connector - to the actor in the given origin location. This doesn't indicate whether or not travel is possible, or where travel goes, or that the actor can tell where the passage goes; this merely indicates whether or not the actor should realize that the passage exists at all.

A closed door, for example, would return true, because even a closed door makes it clear that travel is possible in the direction, even if it's not possible currently. A secret door, on the other hand, would return nil while closed, because it would not be apparent to the actor that the object is a door at all.

isConnectorPassable (origin, traveler): Determine if the travel connection is passable by the given traveler in the current state. For example, a door would return true when open, nil when closed.

This information is intended to help game code probing the structure of the map. This information is NOT used in actor travel; for actor travel, we rely on custom checks in the connector's TravelVia handler to enforce the conditions of travel. Actor travel uses TravelVia customizations rather than this method because that allows better specificity in reporting failures. This method lets game code get at the same information, but in a more coarse-grained fashion.

isConnectorVisibleInDark (origin, actor): Can the given actor see this connector in the dark, looking from the given origin? Returns true if so, nil if not.

This is used to determine if the actor can travel from the given origin via this connector when the actor (in the origin location) is in darkness.

By default, we implement the usual convention, which is that travel from a dark room is possible only when the destination is lit. If we can't determine our destination, we will assume that the connector is not visible.

noteTraversal (traveler): Note that the connector is being traversed. This is invoked just before the traveler is moved; this notification is fired after the other travel-related notifications (beforeTravel, actorTravel, travelerLeaving). This is a good place to display any special messages describing what happens during the travel, because any messages displayed here will come after any messages related to reactions from other objects. (By default this method does nothing, and can be freely overridden with your own code; note, however, that it is overridden by the library in TravelWithMessage, and hence in the subclasses of [TravelWithMessage](#), such as [TravelMessage](#), [NoTravelMessage](#), and [FakeConnector](#), as well).

rememberTravel (origin, actor, dest): Service routine: add a memory of a successful traversal of a travel connector. If we have a travel memory table, we'll add the traversal to the table, so that we can find it later. This is called from Traveler.travelerTravelTo() on successful travel. We're called for each actor participating in the travel.

2.33. Room Methods and Properties

2.33.1. roomXxxxAction

We have now explored all the main types of Room and TravelConnector in the standard library that an author is likely to use (we have not included classes such as BasicLocation, Passage and Stairway that are unlikely to be used directly, since one would normally use one of their subclasses). But before leaving the topic of rooms it may be worth looking at one or two of the methods and properties that can be overridden on them to customise their behaviour.

We have already seen how to customise the [atmosphereList](#) and [brightness](#) properties, so we shall start with the roomAfterAction and roomBeforeAction methods. These are called on the room object whenever an action is performed within that room, either after or before the action. In addition, the roomBeforeAction can abort an action by calling the exit macro. As ever, this is probably best illustrated by means of an example, which we'll provide by adding roomBeforeAction and roomAfterAction methods to the Cabin class:

```
class Cabin : ShipboardRoom, Room
    roomBeforeAction()
    {
        if(gActionIs(Jump))
        {
            "{You/he} had better not try jumping here, {you/he} might hit
            {your} head on the deck beams. ";
            exit;
        }
    }
    roomAfterAction
    {
        if(gActionIn(Look, Examine))
        {
            "\nThe ship creaks ominously.\n";
        }
    }
};
```

The gActionIs macro tests for the action that is either about to be performed in the room. If the Player Character attempts to jump in the cabin he or she is warned that doing so might result in a collision of head and deck beams and the action is aborted. We use the parameter substitution syntax ({You/he} etc.) to deal with the possibility that an NPC is made to jump in the cabin. The gActionIn macro tests for an action matching any of the actions in a list; we use it in roomAfterAction, which tests for either a LOOK or an EXAMINE command being performed, and then displays a message about the ship creaking after the results of the LOOK or EXAMINE. This example is somewhat contrived, and one would probably use some other method to describe the creaking of the ship (although this one may well do well enough) or else have roomAfterAction call the doScript method of an EventList object to vary the message displayed, but the example will suffice to give the general idea. If you like, you can compile and run the game to see what happens in a cabin when you try to JUMP, LOOK or EXAMINE there.

Perhaps the most important point to remember here is to use the roomAfterAction and roomBeforeAction methods for this type of effect; using afterAction or beforeAction on a Room doesn't work.

Note also that in the above code snippet I've put brackets after roomBeforeAction() but not after roomAfterAction. Where a method takes no parameters either is correct (brackets or no brackets) and it makes no difference which you use.

2.33.2. roomParts

For a normal Room the library supplies a defaultFloor, defaultCeiling and four defaultWalls which provide a default "You see nothing special about the floor/ceiling/wall" message if examined. An OutdoorRoom simply has a defaultGround and defaultSky which perform the same function. These objects are listed in the roomParts property of the respective classes, so that they are always available to be examined in any Room or OutdoorRoom. This property can always be overridden, however, if you want more specific or appropriate roomParts for individual Rooms or classes of Room. For example, we may define some roomParts more appropriate to the Deck of our ship on its subterranean lake:

```
defaultDeck : Floor 'deck/ground/floor' 'deck'
    "The deck is made of close-fitting wooden planks. "
    putDestMessage = &putDestFloor
;

caveSky : RoomPart 'roof/ceiling' 'ceiling'
    "The dark roof of the cave, a long way up, dimly reflects the
    rippling green light from the lake. "
;

class Deck : ShipboardRoom, OutdoorRoom
    roomParts = [defaultDeck, caveSky]
;
```

Note that rooms should generally have one and only one roomPart that represents the floor of the room, which must be of class Floor; the main exception here is any room that is *meant* to be floorless, such as a FloorlessRoom or a room defined with the Floorless mix-in class. Since we have made the top of the mast a Floorless, Deck, changing the room parts of [Deck](#) leaves the top of the mast with caveSky as its only roomPart, which is, in fact, just what we want. We could have achieved precisely the same result by defining:

```
topOfMast : FloorlessRoom 'Top of Mast' 'the top of the mast'
    "From the top of the mast you can see miles out across the lake to starboard
    and the shore over to port. The deck below looks a sickently long way down. "
    down = mainDeck
    bottomRoom = (mainDeck.destination)
    roomParts = [caveSky]
;
```

The defaultDeck and defaultCeiling will serve well enough for a Cabin, but it is hardly appropriate for a Cabin to have the north, south, east and west walls found by default in a Room, so we need to provide a new set of roomParts:

```
defaultForeBulkhead : RoomPart 'f fore foreward bulkhead/wall*walls' 'foreward bulkhead';

defaultAftBulkhead : RoomPart 'a aft bulkhead/wall*walls' 'aft bulkhead';

defaultPortWall : RoomPart 'p port wall*walls' 'port wall';

defaultStarboardWall : RoomPart 'sb starboard wall*walls' 'starboard wall';

class Cabin : ShipboardRoom, Room
    roomBeforeAction()
    {
        if(gActionIs(Jump))
        {
            "{You/he} had better not try jumping here, {you/he} might hit
            {your} head on the deck beams. ";
            exit;
        }
    }
    roomAfterAction
    {
        if(gActionIs(Look))
        {
            "\nThe ship creaks ominously.\n";
        }
    }
    roomParts = [defaultDeck, defaultCeiling, defaultForeBulkhead, defaultAftBulkhead,
```


TADS 3 Tour Guide

```
defaultPortWall, defaultStarboardWall]
```

```
;
```

The other change made here is to remove the `gActionIs(Examine)` from the `roomAfterAction`, since otherwise the creaking message will mask the default "You see nothing special about it" response to an attempt to examine these default cabin parts.

In the `greatCabin`, however, even some of these specialised `roomParts` may not be entirely appropriate, since the aft bulkhead is taken up with a window and the foreward one may have a special opening revealed by the press of a button. We can thus further customise the `roomParts` for this particular room:

```
greatCabinForeBulkhead : defaultForeBulkhead
  desc = "The foreward bulkhead is made of polished oak planks.
    <<bulkheadDoor.isOpen ? bulkheadDoor.desc : nil>> "
;

greatCabinAftBulkhead : defaultAftBulkhead
  desc = "The aft wall of the cabin is pierced by a series of windows across
    most of its width. "
;

greatCabin : Cabin 'Great Cabin' 'the great cabin'
  "The great cabin occupies the entire width of the ship at the stern. The stern
  windows aft look out over the water, while there is a solid wooden bulkhead
  foreward. The main piece of furniture is a sturdy wooden desk, while a flight of
  steps leads up to the deck above. "
  up = cabinSteps
  fore = bulkheadDoor
  roomParts = static inherited - defaultAftBulkhead - defaultForeBulkhead
    + greatCabinAftBulkhead + greatCabinForeBulkhead
;
```

There's a couple of points to note here: the first is that we can make our specialised room parts inherit from their corresponding default *objects*; this avoids the need to specify the vocabulary and name properties all over again. The second is the use of the `static` and `inherited` keywords to adjust the list of `roomParts` from that specified in the `Cabin` class (rather than having to list the whole lot again). We use `static` since the list of `roomParts` will never be changed during the game, so the expression that follows the `static` keyword can be resolved at compile time rather than being evaluated when the game is run.

There's also a couple of points to note about `roomParts` in general. The main one is that the library apparently expects them to remain fixed throughout the duration of the game, which will normally be the case (most rooms with four walls, a floor and a ceiling tend to keep them). There may, however, be odd occasions when you want to change the list of room parts in a particular location during the course of a game: perhaps you blow a hole in one of the walls, or the ceiling collapses, or the floor gives way. The thing to note there is that if you want to remove a room part from a room during the course of the game you need to remove it *both* from the location's `roomParts` list *and* its `contents` list. For example, if the main bathroom's ceiling is blown away in a hurricane, you'd need to write something like:

```
mainBathroom.roomParts -= defaultCeiling;
mainBathroom.contents -= defaultCeiling;
```

Similarly, if you want to *add* a room part to a location dynamically during the course of a game you'll need to add it both to the location's `roomParts` list and to its `contents` list. The reason for this is that `Room`'s `initializeThing` method appends the location's `roomParts` list to its `contents` list, but the library provides no automatic means of maintaining this link thereafter.

Where a room has custom room part that you want to add or remove dynamically, prior to TADS 3.0.9 it might be simpler not to include it in the `roomParts` list at all. The alternative would be simply to place it in its room using the location property in the normal way, for example:

```
bathroomCeiling: RoomPart 'ceiling' 'ceiling' @mainBathroom
  "It's full of cracks and looks like it wouldn't take much to make it collapse altogether.
"
;
```

Then, when the bathroom ceiling finally does collapse, all you need to write is:

```
bathroomCeiling.moveInto(nil);
```

TADS 3 Tour Guide

Of course, if you do this, you need to remember to exclude defaultCeiling from the list of roomParts when you define mainBathroom. A further consideration is that although it's reasonably easy to cope with custom walls and ceilings this way, custom floors are a different matter: the library expects the floor (or ground) of a room to be among its roomParts, and although this behaviour can be overridden on a given room (e.g. by overriding its roomFloor property), it's probably simplest to stick to this rule.

TADS 3.0.9 added two new RoomPart methods, moveIntoAdd(room) and moveOutOf(room), which provide a possibly neater alternative. You can now include bathroomCeiling in the bathroom's roomParts property in the normal way, and when the ceiling collapses simply call:

```
bathroomCeiling.moveOutOf(bathroom);
```

If the ceiling were subsequently repaired you could reverse this by calling:

```
bathroomCeiling.moveIntoAdd(bathroom);
```

This leads on to a more general point: where you want to use customised walls and ceilings (and possibly even floors) in a given location, it's always possible to by-pass the roomParts mechanism altogether and simply make your customised room parts ordinary Fixtures located in the room. If you do this, it's probably better to use the RoomPart class for them than the Fixture class (since the RoomPart class contains some specializations that make better sense for things like walls and ceilings), and you still have to remember to remove the default room part equivalents from the location's roomParts list when you define the room, or you could end up with, say, two west walls, the default one and your custom one.

For this reason, it's probably easier to get into the habit of always putting room parts - even custom ones - into their location's roomParts list. This way you're much more likely to remember to remove the corresponding default room part, and you also make sure you take advantage of the library's specialized handling of room parts. For example, if you define a custom ceiling for one location, and then find it would suit another just as well, it's probably easier to add it to the roomParts list of both locations than to make into a [Multinstance](#) object.

2.33.3. cannotGoThatWay

BasicLocation.cannotGoThatWay is called whenever an actor (usually the PC) attempts travel in a direction that is not currently available (except in the dark, when [cannotGoThatWayInDark](#) is used instead). By default this simply displays a message saying you can't go that way, and listing the exits that are available from the current location. There may be occasions, however, when you'd like a different message displayed.

Consider our shipboard locations, the Decks and Cabins. Just as a shipboard direction such as PORT or AFT is that meaningful on dry land, we may feel that compass directions such as NORTH or SW are not that relevant to moving around a ship. We could therefore override the cannotGoThatWay method of Deck to display a more appropriate message when travel in a compass direction is attempted:

```
class Deck : ShipboardRoom, OutdoorRoom
    roomParts = [defaultDeck, caveSky]
    cannotGoThatWay()
    {
        if(gAction.parentAction.dirMatch.dir.ofKind(CompassDirection))
            "Compass directions aren't that useful for getting about ship;
            try fore, aft, port and starboard instead. ";
        else
            inherited;
    }
;
```

The complicated part here is getting at what kind of direction the player typed, but the above seems to work. The easy part is extending this behaviour to the Cabin class; simply make Cabin inherit from Deck instead of from ShipboardRoom and Room. Since Deck inherits from ShipboardRoom and OutdoorRoom, the only difference between Room and OutdoorRoom is the list of roomParts, and Cabin overwrites roomParts anyway, this change is perfectly safe.

More generally, if you want to provide custom "Can't go that way" in a number of different locations, you may just need to provide a cannotGoThatWayMsg:

TADS 3 Tour Guide

```
squareCave : DarkRoom 'Square Cave' 'the square cave'
    "This capacious cave is unnaturally square, suggesting that it has been
    artificially hewn out of the rock, an impression further enhanced by
    the carefully-constructed ashlar archway to the west. "
    west = mainCave
    out asExit(west)
    cannotGoThatWayMsg = 'You can\'t go through solid rock! '
;
```

Where the property is not overridden, however, the default "You can't go that way" message will be displayed as before.

2.33.4. cannotGoThatWayInDark

By default, the `cannotGoThatWayInDark` method of a `Room` (or `BasicLocation`) displays a message to the effect that you can't see where you're going in the dark. We might want to change that in particular cases. For example, the description of the `crewQuarters` suggests that there's a ladder leading down into the hold. If the player character goes blundering about the `crewQuarters` in the dark there's always the danger that he or she will end up falling down the ladder and kill themselves. To be fair, though, we may first want to warn the player character that wandering around in the dark could prove dangerous, so we might do it this way:

```
crewQuarters : DarkCabin 'Crew Quarters' 'the crew quarters'
    "The crew quarters seem largely deserted. There's an exit back aft and a
    ladder leading down into the hold. "
    down = holdLadderDown
    aft = greatCabin
    cannotGoThatWayInDark()
    {
        darkEvents.doScript();
    }
    darkEvents : StopEventList
    {
        [
            'Blundering about a ship in the dark could prove dangerous. ',
            new function
            {
                "Blundering around in the dark you fall down a ladder into the hold
                and break your neck. ";
                endGame(ftDeath);
            }
        ]
    }
;
```

Note that the `endGame` function isn't part of the standard library; it's used here as a convenient wrapper for the `finishGameMsg` function., so the next job is to define this function:

```
function endGame(msg)
{
    finishGameMsg(msg, [finishOptionUndo, finishOptionFullScore]);
}
```

The purpose is to avoid having to specify the same options (`finishOptionUndo`, `finishOptionFullScore`) each time we want to end the game. The call to `endGame(ftDeath)` prints a "YOU HAVE DIED" message and ends the game with a set of options such as UNDO, RESTART, FULL SCORE or QUIT; `endGame(ftVictory)` would do the same but with the message "YOU HAVE WON". You can also supply your own message by supplying a single-quoted string as the `msg` argument, e.g. `endGame("YOU HAVE FAILED DISMALLY")`.

Note also that there is one situation that the code above does not handle, namely if the player tries to go DOWN from the `crewQuarters`. We'll fix that next by overriding [roomDarkTravel](#).

2.33.5. roomDarkTravel

`BasicLocation.roomDarkTravel()` defines what happens if we try to move from the current location when it's dark to another dark location. By default, it simply displays the same message as [cannotGoThatWayInDark](#) and then uses `exit` to cancel the movement action. In most cases you'll probably want to keep both methods appearing to do the same thing (unless you want to allow travel from one dark location to another), so that the player is given no indication in the dark whether a given direction is valid for travel or not. In this case we could simply override `roomDarkTravel` to call `cannotGoThatWayInDark` and then `exit`:

```
crewQuarters : DarkCabin 'Crew Quarters' 'the crew quarters'
"The crew quarters seem largely deserted. There's an exit back aft and a
ladder leading down into the hold. "
down = holdLadderDown
aft = greatCabin
cannotGoThatWayInDark()
{
    darkEvents.doScript();
}
roomDarkTravel(actor)
{
    cannotGoThatWayInDark;
    exit;
}
darkEvents : StopEventList
{
    [
        'Blundering about a ship in the dark could prove dangerous. ',
        new function
        {
            "Blundering around in the dark you fall down a ladder into the hold
            and break your neck. ";
            endGame(ftDeath);
        }
    ]
}
;
```

In this case the player only gets one warning; if the PC leaves the `crewQuarters` aft to the `greatCabin` after making one false step in the dark, the next false step in `crewQuarters` in the dark will kill the PC off. This may be what you want, but we'll try changing it next using [enteringRoom](#).

2.33.6. enteringRoom

It is sometimes useful to have something happen each time an actor arrives in a room. For example, we may want to reset the state of the `darkEvents` `StopEventList` each time the player character enters the `crewQuarters` so that there is always one warning about blundering about in the dark before the PC falls down the ladder and dies. This can be achieved by overriding `enteringRoom`:

```
crewQuarters : DarkCabin 'Crew Quarters' 'the crew quarters'
"The crew quarters seem largely deserted. There's an exit back aft and a
ladder leading down into the hold. "
down = holdLadderDown
aft = greatCabin
cannotGoThatWayInDark()
{
    darkEvents.doScript();
}
roomDarkTravel(actor)
{
    cannotGoThatWayInDark;
    exit;
}
darkEvents : StopEventList
{
    [
        'Blundering about a ship in the dark could prove dangerous. ',
        new function
        {
            "Blundering around in the dark you fall down a ladder into the hold
            and break your neck. ";
            endGame(ftDeath);
        }
    ]
}
;
```

TADS 3 Tour Guide

```
[
  'Blundering about a ship in the dark could prove dangerous. ',
  new function
  {
    "Blundering around in the dark you fall down a ladder into the hold
    and break your neck. ";
    endGame(ftDeath);
  }
]
}
enteringRoom (traveler)
{
  darkEvents.curScriptState = 1;
}
;
```

The **enteringRoom** method is a convenience hook that is called from **travelerArriving**, which performs some significant processing of its own and which uses a longer parameter list. By default, the library method **enteringRoom** does nothing, so that we do not need to call **inherited**. Without the **enteringRoom** method we should instead have had to write:

```
travelerArriving (traveler, origin, connector, backConnector)
{
  darkEvents.curScriptState = 1;
  inherited (traveler, origin, connector, backConnector);
}
```

There is also a corresponding **leavingRoom(traveler)** method that can be used to execute custom code when a traveler is about to leave a room.

2.33.7. inRoomName

The **inRoomName(pov)** method is used to define how a room should be named when listing its contents from the point of view of another location. The method should return a single-quoted string. For further explanation and an example, see [DistanceConnector](#).

3. NonPortables

3.1. NonPortable Introduction

Most of the items we have added to the game so far have been NonPortables - that is objects that cannot be picked up and moved around - but that is because they have mainly been various types of room and passage. In this section we shall take a look at the principal kinds of NonPortable object one might use as part of the contents of a Room, giving a few examples to start furnishing the rooms we have created so far.

One common feature of NonPortable objects to be borne in mind is that, by default, they are not shown in listings of the contents of rooms or other objects. This is because they are considered to be permanent features of their location, and should therefore be mentioned in the description of their room or other container, or else given an `initSpecialDesc` or `specialDesc` (which will be listed). This behaviour can be changed by overriding the `isListed`, `isListedInContents`, and `isListedInInventory` properties of a NonPortable object. Note that the fact that a NonPortable is not listed does not of itself make it invisible: it can still be EXAMINED and will respond to other commands directed towards it.

You are not likely to declare an object to be of class NonPortable in your game code, since NonPortable serves principally as a common ancestor class to a number of different classes that are commonly used. A partial tree of NonPortable classes, some of which we have already met, is as follows:

NonPortable

```

Fixture
  Component
    ComplexComponent
  Decoration
    Unthing
  Distant
  Enterable
    EntryPortal
  Exitable
    ExitPortal
  NominalPlatform
  Passage
  Room
  RoomPart
  SecretFixture
Immovable
  Heavy
    TravelPushable

```

A Note on Notation

In what follows we shall specify the room location of objects using the @notation of the Thing template, rather than the + notation, e.g. by writing

```

myThing : Thing 'my thing' 'thing' @outsideCave
  "A poor thing, but mine own. "
;

```

Rather than

```

+ myThing : Thing 'my thing' 'thing'
  "A poor thing, but mine own. "
;

```

Either method is possible in your own code; the reason for doing it this way here is to avoid the need for (and possible confusion arising from) specifying where in existing code these new objects need to be placed. There is also something to be said for specifying the objects in a different part of the code - even a different source file - from the rooms and connectors, since this leaves the basic outline of the map clearer in the room code. The downside is that it may be less immediately apparent how objects and rooms relate to each other.

3.2. Fixture

The Fixture class is for items that are quite evidently fixed in place within their locations. Unless a Fixture is given an `initSpecialDesc` or `specialDesc` property, it is not normally listed as part of the contents of a room, since it is assumed that some reference will have been made to it in the description of the room. Some such Fixtures have already been implemented as Passage objects; now we'll add a few others.

For example, the description of `mainCave` refers to a torch fixed to the wall, so we might implement it as a Fixture (although later we shall also need to make it a `FireSource`):

```
Fixture 'torch' 'torch' @mainCave
    "The torch, which is fixed firmly to wall by a steel bracket, is blazing merrily,
    its flames casting a bright but flickering light over the cave. "
    cannotTakeMsg = 'It\'s fixed to the wall. '
;
```

Note that we have overridden `cannotTakeMsg` to give a slightly more meaningful response than the default when the player attempts to take the torch. It would also be possible to override the `cannotMoveMsg` and `cannotPutMsg` in a similar way. If any of these properties is overridden it should be with a *single*-quoted string (or a property pointer) and never with a double-quoted string.

The description of the `Quarterdeck` likewise refers to a deck rail, which we can implement thus:

```
Fixture 'wooden (deck) rail' 'deck rail' @quarterDeck
    "The wooden deck rail runs along the forward edge of the Quarterdeck,
    separating it from the main deck, although it is possible to get round
    the rail either to starboard or port to go foreward. A large wooden
    panel is fixed to the centre of the rail. "
;
```

In neither case is it necessary to give names to these objects, since they will not be referred to elsewhere in code (though this may not always be the case with Fixtures). Note the use of the 'weak tokens' syntax in the vocabulary for the rail; this allows players to refer to it as a 'wooden deck rail' without its answering to 'deck' alone.

3.3. CustomFixture

A `CustomFixture` is simply a fixture that uses the same custom message for taking, moving, and putting. In many cases, it's useful to customize the message for a fixture, using the same custom message for all sorts of moving. Just override **`cannotTakeMsg`**, and the other messages will copy it.

We haven't yet reached the point in our game where we need a `CustomFixture`, but we'll eventually use one to represent the pillars in a [temple](#).

See also the similar but subtly different [CustomImmovable](#).

3.4. Decoration

The normal purpose of a `Decoration` object is to provide a description of an object mentioned in a room description or other object description, when the object is of no real importance to the game but ought to be implemented for the sake of completeness. For example, consider the following transcript:

> LOOK

Entrance Cave

This large cave forms the main entrance to the whole underground complex.

TADS 3 Tour Guide

A red sign on one wall points to the north; next to it is a blue sign.

A sturdy steel ladder leads down through a large round hole in the floor, and a narrow ledge is carved into one wall.

>X RED SIGN

You see no red sign here.

Even if the red sign is of no importance to the game, this is frustrating to the player. A Decoration object gets round this by providing something that produces a description in response to an EXAMINE command and a message like 'The red sign is not important.' in response to any other action attempted upon it. We could thus implement the two signs mentioned in the entranceCave as follows:

```
Decoration 'red sign*signs' 'red sign' @entranceCave
  "\n<FONT COLOR=WHITE BGCOLOR=RED FACE='Tads-Typewriter'>WAY OUT -></FONT>\n"
  dobjFor (Read) asDobjFor (Examine)
;

Decoration 'blue sign*signs' 'blue sign' @entranceCave
  "\n<FONT BGCOLOR=BLUE COLOR=WHITE FACE='TADS-Typewriter'>
    WELCOME TO&nbsp;THE\NEERHTSDAT CAVES</FONT>\n"
  dobjFor (Read) asDobjFor (Examine)
;
```

Note that we have remapped READ to EXAMINE for these signs since a player might quite reasonably expect to be able to read a sign as well as examine it. Note also the **signs* syntax in the vocabulary of these objects. Any word after an asterisk (*) in an object's vocabulary is considered a plural (or other collective noun) for that object. In this instance this allows a player type X SIGNS or READ SIGNS and have both signs described by the same command.

According to the room description of mainCave, the torch is simply fixed to the wall. If the player examines the torch however, he or she is told that the torch is fixed to the wall by means of a steel bracket. Players are not meant to interact with the bracket in any other way, but since they may try to, it is a good candidate for a Decoration object.

```
Decoration 'steel bracket' 'steel bracket' @mainCave
  "The steel bracket is fixed securely to the wall; there doesn't appear to be
    any way it could be detached. "
;
```

Included in the description of longCave is the notice that "Some words have been crudely scratched on the south wall." A Decoration object may well be just the thing to represent these words, but this requires a little more thought. By default if we try to do anything to these words but EXAMINE them, the game will report "The words aren't important." This may not be the message we want to convey here, since what the writing on the wall says may actually have some significance. To deal with this need we need to override the Decoration's **notImportantMsg** property with something more appropriate. Moreover, it would be reasonable for the player to attempt to READ the words as well as EXAMINE them; as in the case of the two signs in the Entrance Cave, we want READ to be treated like EXAMINE rather than displaying whatever we put into notImportantMsg, so once again we need dobjFor(Read) asDobjFor(Examine). There is one further complication: the writing is described as being scratched on the south wall, so it ought to be described if the player examines the south wall; to achieve this we need to associate the words with the south wall of the cave:

```
longCaveWords : Decoration 'words/writing' 'words' @longCave
  "The writing on the wall declares:\b
    <q>One banana to rule them all\nAnd in the darkness bind them.</q>"
  isPlural = true
  notImportantMsg = 'That\'s not the sort of thing you can do to them. '
  dobjFor (Read) asDobjFor (Examine)
  initNominalRoomPartLocation = defaultSouthWall
;
```

The last line of this definition (excluding the final semicolon) tells the system that the longCaveWords are nominally on the south wall. This allows the player to EXAMINE WORDS ON SOUTH WALL as well as EXAMINE WORDS and have the description displayed. It also causes the words to be mentioned when the player types EXAMINE SOUTH WALL (note that prior to version 3.0.9 it would also have been necessary to override isListedInRoomPart to achieve this effect, but this is no longer necessary in 3.0.9).

A further refinement offered in version 3.0.9 is the new mix-in class RoomPartItem. This allows us to set up an item that displays its specialDesc (or initSpecialDesc) only when the room part to which its nominally attached is examined. This is useful for objects such as doors and windows that might already be included in the general room description,

TADS 3 Tour Guide

or for objects that are not worth listing in their own right but which are worth a mention when the room part to which they are attached is examined. The advantage of using `specialDesc` (or `initSpecialDesc`) for this purpose is that we can customise the way the fixture is described, instead of producing something a bit ungainly like, "On the north wall is a red door. " As an example, we might further customise the bracket object so that when the north wall of the cave is examined we see "A steel bracket containing a flaming torch is attached to the wall. ":

```
bracket : RoomPartItem, PermanentAttachment, Decoration 'steel bracket' 'steel bracket'
    @mainCave
    "The steel bracket is fixed securely to the wall; there doesn't appear to be
    any way it could be detached. "
    specialNominalRoomPartLocation = defaultNorthWall
    specialDesc = "A steel bracket containing a flaming torch is fixed to the wall. "
;
```

Note that in this case, since the bracket will never move, it doesn't matter whether we use `specialNominalRoomPartLocation` and `specialDesc`, or `initNominalRoomPartLocation` and `initSpecialDesc`, as long as we use one pair or the other and don't try to mix them. If the bracket could be removed from the wall, we'd probably want to use `initNominalRoomPartLocation` and `initSpecialDesc`.

Finally, a simple example of a `Decoration` would be the lake as seen from the shore. There seems little reason why the lake should look any different from `lakeRoom` and `pathEnd`; rather than define the same decoration twice, we can thus take a shortcut by making it a `MultiLoc`; strictly speaking, it should perhaps be a `MultiInstance`, but in this case no harm will come of using `MultiLoc` and it's slightly simpler.

```
MultiLoc, Decoration 'great (giant) underground lake/water' 'lake'
    "The lake, which stretches as far south as you can
    see, looks almost as flat as a millpond, although the occasional
    ripple runs across its surface. It is also strikingly
    phosphorescent, casting an eerie green glow over the whole
    vast cavern. "
    locationList = [lakeRoom, pathEnd]
;
```

The point to bear in mind here is that a [MultiLoc](#) represents a single physical object present in more than one location, and one that is sufficiently small that, for example, if it is lit in one location it is lit in all and if something is put in it in one location it can be retrieved from it in another. The lake meets the first of these conditions, but not the second. Because, however, it's a `Decoration`, the only relevant consideration is lighting. If one part of the lake might be in darkness while another was lit, it would be inappropriate to use a `MultiLoc` to represent it (since what was meant to be the dark part of the lake would appear as lit). In this game, however, all parts of the lake will be permanently lit, so it's safe to make it a `MultiLoc`.

The general principle here is that it's safe to make a `Decoration` a `MultiLoc` if and only if the lighting conditions are always the same in all the locations where the `Decoration` exists (it's fine if all the lighting conditions change simultaneously, but they must always be the same in each location at any one time). If this condition is not met, use a [MultiInstance](#) instead.

3.5. Distant

A `Distant` is a special type of [Decoration](#) that represents an object that's too far away to interact with, perhaps an object that's in another location. The lake as seen from the top of the mast might come into this category:

```
Distant 'great underground lake' 'lake' @topOfMast
    "The lake stretches out to starboard as far as the eye can see; it looks as
    calm and flat as a millpond. "
;
```

The shore as seen from the same place might also come into this category. Since eventually the ship will move around the description must either be studiously vague or else vary according to the location of the ship:

```
Distant 'shore' 'shore' @topOfMast
    desc()
    {
```

TADS 3 Tour Guide

```
switch(ship.location)
{
  case lakeRoom:
    "The shore to port is a narrow strip of land bounded by the wall of the
    cave, through which a doorway leads to the north. ";
    break;
  default:
    "The shore is directly on the port side of the ship. ";
}
;
;
```

Clearly, we should come back and expand the desc method once we've implemented more of the locations the ship can go to. The points to note here are (1) that desc() can be a method (in which case we need to name it explicitly, not via the template) and (2) to remember to use the break statement in each branch of the switch statement where we don't want fall-through.

3.6. Unthing

An Unthing is a special kind of Decoration used to represent something that *isn't* present, but to which the player might try to refer; it then displays its notHereMsg to explain why it isn't there. The most common use for an Unthing is to represent the absence of something that has just disappeared. For example suppose we plant what appears to be treasure in mainCave, but have it disappear when the player attempts to take it. We might then move an Unthing into its place to describe its absence if the player continues to refer to it:

```
fakeTreasure : Thing 'huge great golden gold banana/treasure'
  'golden banana' @mainCave
  "It's a fantastic treasure, over two feet long, and by the look of it, solid
  gold. It must be worth an absolute fortune!"
  initSpecialDesc = "A huge treasure - a great golden banana - lies on the ground. "
  dobjFor(Take)
  {
    action()
    {
      "All that glisters is not gold, and as you try to take the great golden
      banana it crumbles into dust and vanishes away. ";
      noTreasure.moveInto(location);
      moveInto(nil);
    }
  }
  getFacets() { return [noTreasure]; }
;

noTreasure : Unthing 'huge great golden gold treasure/banana/dust' 'golden banana'
  'The illusory golden banana vanished into fine dust that is no
  longer visible. '
;
;
```

Note the use of getFacets on fakeTreasure, so that if a player types TAKE BANANA followed by, say, X IT, the parser will know that IT now refers to the noTreasure object that's just been substituted for the fakeTreasure. In this case there's no need to add a getFacets method to noTreasure, since the fakeTreasure will never reappear to be referred to as IT. Note also the range of vocabulary words we have given to both objects, and that we added 'dust' to the list of words by which the noTreasure object can be referred to.

Note that the third property we have defined on Unthing is *single-quoted string*, not a double-quoted string. This is because there is a special [Unthing template](#) which puts the notHereMsg instead of desc in third place. We don't want to define desc on an Unthing, because it's not generally useful, we just want to define the notHereMsg which will be used for any command that tries to interact with the Unthing. The above definition of noTreasure is equivalent to:

```
noTreasure : Unthing 'huge great golden gold treasure/banana/dust' 'golden banana'
  notHereMsg = 'The illusory golden banana vanished into fine dust that is no
  longer visible. '
;
;
```

TADS 3 Tour Guide

Or to:

```
noTreasure : Unthing
  vocabWords = 'huge great golden gold treasure/banana/dust'
  name = 'golden banana'
  notHereMsg = 'The illusory golden banana vanished into fine dust that is no
    longer visible. '
;
```

3.7. Immovable

An Immovable object is one that can't be moved but isn't obviously fixed in place. The practical difference between a [Fixture](#) and an Immovable is that moving the former is forbidden in the verify method, while moving the latter is disallowed in the action method.

The messages that are displayed when the player attempts to TAKE, PUT or otherwise MOVE (e.g. PUSH or PULL) an Immovable can be changed by overriding cannotTakeMsg, cannotPutMsg and cannotMoveMsg respectively.

A simple Immovable would be something like a piece of furniture that the player's not allowed to take or move. However, we'll make our example a bit more interesting than that: we'll put a rug in the roundCave that starts by covering the hole in the floor. The player cannot take the rug but he or she can pull it (once only) to reveal the hole beneath. Later we'll also hide a key under this rug:

```
rug : Immovable 'large rectangular chinese rug/pattern/leaves/dragons' 'Chinese rug'
  @roundCave
  "The rectangular rug is patterned in pastel colours, mainly turquoise round the
    edge and principally golds and browns within. The patterns consists mainly
    of leaves and dragons. "
  initSpecialDesc = "A Chinese rug covers the centre of the floor. "
  specialDesc = "The Chinese rug has been pulled over to one side of the cave. "
  cannotTakeMsg = 'You probably could roll the carpet up and drag it around,
    but you don\'t want to be encumbered with it. '
  dobjFor(Pull)
  {
    action()
    {
      if(moved)
        "You can't pull the rug any further, it's already at the edge of the cave. ";
      else
      {
        "Pulling the rug over to the edge of the cave reveals a square hole in the floor. ";
        moved = true;
      }
    }
  }
;
```

There a few things to note here. First, we have used the **moved** property of the rug to determine whether or not the rug has been pulled to one side. This isn't its normal function, since normally moved is used to track whether an object has moved into another location. However, it's convenient here, both because we don't need rug.moved for any other purpose and also because setting moved = true when the rug has been pulled also means that thereafter the specialDesc will be displayed in place of the initSpecialDesc, which happens to be just what we want (since it describes the changed state of the carpet). We have overridden cannotTakeMsg to provide a custom response, and, more importantly, we have overridden the dobjFor(Pull) handling to allow the rug to be pulled a single time to reveal the hole.

This does, of course, require some change to the definition of the hole object so that it appears and can be traversed only when the rug has been pulled aside. The easiest way to achieve this is to change it from a ThroughPassage to a HiddenDoor and to set its isOpen property to rug.moved (since moving this rug effectively opens this previously hidden passage). We also need to change the room description of roundCave so that the hole is mentioned only when the rug has been pulled:

TADS 3 Tour Guide

```
roundCave : DarkRoom 'Round Cave' 'the round cave'
    "This round, rocky cave has a narrow exit to the east<<rug.moved ?
    ' and a strange square hole in the floor' : nil>> . "
    east = mainCave
    down = squareHole
;

+ squareHole : TravelWithMessage, HiddenDoor 'square hole/chute' 'square Hole'
    "The hole is just about large enough for one person to fit through. A glint
    of something metallic can be seen just through the hole. "
    travelDesc = "You find yourself sliding down a long, slippery metal chute;
    After a short ride you are ejected into another cave. "
    isOpen = (rug.moved)
;
```

3.8. CustomImmovable

A CustomImmovable is an Immovable that uses the same custom message for taking, moving, and putting. In many cases, it's useful to customize the message for an immovable, using the same custom message for all sorts of moving. Just override **cannotTakeMsg**, and the other messages will copy it.

At first sight this makes a CustomImmovable look identical in function to a [CustomFixture](#); there is, however, a subtle difference. This is, of course, the same as the difference between an Immovable and a Fixture, namely that while the library regards an attempt to move, push or take a Fixture as illogical (i.e. ruled out in the verify method), it merely disallows taking an Immovable (in the action method). The main practical effect of this is that a CustomFixture will not be considered as a possible candidate for a move, take or push action in disambiguation, while a CustomImmovable will. CustomFixture should therefore be used for things that obviously can't be moved around (like pillars in a temple), while CustomImmovable should be used for things that perhaps could be taken, but in fact cannot be (like the carpet in the [Immovable](#) example, which could just as well have been a CustomImmovable). We'll give another example of a CustomImmovable [later](#).

3.9. Heavy

A Heavy object is one that is too heavy for the player character to lift or move, such a piece of heavy furniture:

```
cabinDesk : Heavy 'large solid oak desk' 'desk' @greatCabin
    "It's a large, solid oak desk. A button is fixed underneath it. "
;
```

We shall be doing more things with this desk in due course.

3.10. Component

As its name suggests, a Component is something that is part of something else. It need not be fixed within a particular room location, since it could be part of a portable object, a button on a mobile device, for example, but it cannot be detached from its immediate parent, and wherever its parent goes, it goes with it. A button on a stationery device equally qualifies, however, so we can now move the button that was defined in greatCabin to a more appropriate location (just after the [desk](#) defined above), and change it from a Fixture to a Component:

```
+ Button, Component 'small brown button' 'small brown button'
    "The small brown button is fixed to the underside of the desk. "
    dobjFor(Push)
    {
        action()
    {
        "There's a sharp <i>click</i>, and a section of the foreward bulkhead slides
```

TADS 3 Tour Guide

```
<<bulkheadDoor.isOpen ? 'closed' : 'open'>>. ";
bulkheadDoor.makeOpen(!bulkheadDoor.isOpen);
}
}
;
```

As yet we have not implemented any portable objects to which a component might be attached, but we have referred to a panel mounted on the deck rail, so we can follow the definition of the deck rail object immediately with:

```
+ Component 'large wooden panel' 'panel'
  "The panel seems to have something to do with sailing the ship. A wheel and a lever
  are mounted on it, and between them is a hexagonal aperture. "
;
```

The panel refers to a wheel, a lever and a hexagonal aperture, all of which will be its components; but we are not in a position to implement any of these just yet.

4. Things

4.1. Thing - Introduction

The Thing class is important in the TADS3 library for two reasons: (1) because it is the class used for all sorts of portable objects the player may interact with and (2) because it is the ancestor class for anything that represents a physical object in game (included those that are [non-portable](#) and some that are intangible). In the present chapter we shall concentrate principally on the first use of Thing - as a class in its own right - but because so many classes inherit (directly or indirectly) from Thing, much of what we say about the properties and methods of Thing will be equally applicable to classes that inherit from Thing.

The properties and methods of Thing we shall be going on to discuss (or at least, exemplify) include:

[brightness](#)
[bulk](#)
[canBeTouchedBy](#)
[desc](#)
[described](#)
[disambigName](#)
[distantInitSpecialDesc](#)
[feelDesc](#)
[globalParamName](#)
[initSpecialDesc](#)
[initDesc](#)
[isEquivalent](#)
[isHeldBy](#)
[isKnown](#)
[location](#)
[material](#)
[moved](#)
[name](#)
[remoteInitSpecialDesc](#)
[seen](#)
[sightSize](#)
[soundSize](#)
[smellDesc](#)
[specialDesc](#)
[tasteDesc](#)
[throwTargetCatch](#)
[useSpecialDesc](#)
[vocabWords](#)
[weight](#)

In the present chapter we shall discuss only the simplest and most common of these, since some of the others will only become relevant in the light of other classes and concepts we haven't covered yet.

There are also one or two subclasses of Thing that are both so straightforward and so miscellaneous they may as well be dealt with in this chapter, namely:

[Food](#)
[Readable](#)
[Wearable](#)

4.2. Thing - The Basics

The basic properties that apply to almost all Thing objects (and objects using many of the classes inheriting from Thing) are vocabWords, name, location, and desc. These are so common the standard [Thing template](#) allows them to be defined without naming them, thus:

```
myObject : Thing 'vocabWords_' 'name' @location
          "desc"
;
```

TADS 3 Tour Guide

And for the most basic portable objects, this type of definition will often suffice without the need to define any other properties or methods. For example, we shall leave a coin for the player to find in the longCave room (using the [Thing template](#)):

```
brassCoin : Thing '(small) brass coin/groat*coins' 'small brass coin' @longCave
    "On the obverse is the head of King Freddie the Fat, and on the reverse
      is stamped ONE GROAT. "
;
```

By now, most of these properties should be familiar. The **desc** (description) property is what is displayed in response to an EXAMINE command; the only real complication is that you may sometimes want to define desc as a *method*, in which case it must be explicitly defined as a named method outside the template.

The **name** property is the what will normally appear when the object is listed in the contents of rooms, containers or inventory, or when the parser needs to refer to the object (E.g. "Which coin do you mean, the brass coin or the gold coin?").

For a Thing the **location** is normally the object's physical container, which may be a room, an actor (including the Player Character) who is carrying or wearing the object, or some other form of physical container (such as a jar or the top surface of a table). The location can also be specified by using the + notation; e.g. to put the coin in longCave we could have written

```
longCave : DarkRoom 'Long Cave' ...
...
;

+ brassCoin : Thing '(small) brass coin/groat*coins' 'small brass coin'
    "On the obverse is the head of King Freddie the Fat, and on the reverse
      is stamped ONE GROAT. "
;
```

Note that if both the + notation and the @location notation are used on the same object, the + notation takes precedence. But if the + notation is used with an explicit setting of the location property, the explicitly named location property takes precedence. For example, in the case of the brassCoin with the + notation, if I added @entranceCave to the object definition after 'brass coin' the coin would remain in longCave, but if I added location=entranceCave the brass coin would start life in the entranceCave, despite the + property. This can sometimes be useful if you have a sequence of objects nested within one another using the + notation and you want to define an object that doesn't belong in the containment hierarchy amongst those that do.

Note also that if location is an expression or method, it must be explicitly defined as a named property outside the template, e.g. location = (ship.location)

The [vocabWords](#) property is perhaps the most complex of the four, so we shall discuss it in a separate section.

4.3. vocabWords

The vocabWords property defines the vocabulary with which the player can refer to the object. The definition of brassCoin is

```
brassCoin : Thing '(small) brass coin/groat*coins' 'small brass coin' @longCave
    "On the obverse is the head of King Freddie the Fat, and on the reverse
      is stamped ONE GROAT. "
;
```

In this definition the format of the vocabWords property defined through the template is:

```
(weakToken) adjective noun/noun*plural
```

A weak token is a word that may be included among the words used to identify an object, but which is not sufficient to do so by itself. In this case, for example, the player may call the coin SMALL BRASS COIN or SMALL GROAT or SMALL BRASS and the parser will know what is meant, but the coin will not answer to being referred to simply as SMALL (as in EXAMINE SMALL or TAKE SMALL). Any word (it need not be the first) included in parentheses in the

TADS 3 Tour Guide

vocabWords property of a Thing is a weak token. We have here made SMALL a weak token since it seems too common a word to stand on its own as defining which object is meant.

The functional difference between adjectives and nouns is that any number of the listed adjectives may be used by the player to identify the object, but only one of the nouns (but see below for an exception to this). Thus the player may type X SMALL COIN, or X BRASS COIN or TAKE SMALL BRASS GROAT and the parser will accept all of these as valid references to the coin. However, if the player types X GROAT COIN or X SMALL COIN GROAT this will not be taken as referring to the coin. If you felt GROAT COIN was a valid way of referring to this object you could allow it by adding 'groat' to the list of adjectives as well, i.e.

```
+ brassCoin : Thing '(small) brass groat coin/groat*coins' 'brass coin'
  "On the obverse is the head of King Freddie the Fat, and on the reverse
    is stamped ONE GROAT. "
;
```

Players can then refer to it as a SMALL BRASS GROAT COIN if they so wish.

The plural (anything after the asterisk) can be used to refer to a number of coin objects collectively. For example, if we defined a silver coin and a gold coin, and gave them both a plural of 'coins', then, provided all three were in scope, the word COINS could be used to refer to all three coins at once. For example, X COINS would list a description of all three coins and TAKE COINS would cause the Player Character to pick up all three coins (assuming that TAKE was a valid action for all three coins when the command was issued).

And now for the exception to the rule that an object can only match one noun at a time. On occasion one can have an object that essentially contains two nouns connected by 'of' in its name like 'pile of rubbish' or 'golden banana of discord'. In this case you simply define both nouns in the normal way; for example, for an object that will match 'golden banana of discord' you could define:

```
goldenBanana 'golden banana/discord' 'Golden Banana of Discord'
  "It's golden and banana-shaped. "
;
```

A further complication of the vocabWords property is that you can't usefully change the vocabulary used to refer to an object by a programming statement that manipulates it directly. For example, if you wanted the player to be able to refer to the coin as a groat only after something else had occurred (perhaps his examining the coin) you could not achieve this by writing a statement like:

```
brassCoin.vocabWords += 'groat';
```

Since although this code would execute, it would not have the desired effect. Instead the easiest way to add vocabulary to an object is with the initializeVocabWith() method, which accepts a string argument in the same format as the vocabWords property, so we could write:

```
brassCoin.initializeVocabWith('groat');
```

To add 'groat' as a noun to the brassCoin's vocabulary. Or even

```
brassCoin.initializeVocabWith('little shiny object');
```

To add 'little' and 'shiny' as adjectives and 'object' as a noun.

An alternative is to use cmdDict.addWord(obj, str, voc_prop), e.g. to achieve the same as the previous example:

```
cmdDict.addWord(brassKey, 'little' &adjective);
cmdDict.addWord(brassKey, 'shiny' &adjective);
cmdDict.addWord(brassKey, 'object' &noun);
```

Although this is rather more long-winded. You can use the similar removeWord method to take vocabulary away from an object, which may occasionally be useful. For example, let's suppose that when the coin is first seen lying on the ground it just appears to be a small brassy object. We want it referred to as a small brassy object until it's examined, after which it becomes a small brass coin; at that point we no longer want the vague word 'object' to refer to it, but until then the player can't refer to it as a coin or groat. We can achieve this with the following:-

```
brassCoin : Thing '(small) brassy object' 'small brassy object' @longCave
  "On the obverse is the head of King Freddie the Fat, and on the reverse
    is stamped ONE GROAT. "
  dobjFor (Examine)
```


TADS 3 Tour Guide

```
{
    action()
    {
        inherited;
        changeName();
    }
}
changeName()
{
    name = 'small brass coin';
    cmdDict.removeWord(self, 'object', &noun);
    initializeVocabWith('brass coin/groat*coins');
}
;
```

4.4. initDesc & initSpecialDesc

If the coin starts life lying on the ground as a small brassy object, rather than seeing a description that reads "You see a small brassy object here" it would be nicer if it read something like "A small brassy object lies on the ground in a dim corner of the cave." Likewise, if we examined the coin without first picking it up it would be good if we obtained a vaguer description such as "It looks like it might be a coin of some sort." - after all, the standard description we've given the coin refers to what's on its obverse and its reverse, but how can we see what's on both sides of the coin while it's still lying on the ground?

To achieve this we can use the `initSpecialDesc` and `initDesc` properties. The first of these, `initSpecialDesc`, is what will be displayed in a room or contents listing before the object has been moved (while its `moved` property is nil); `initDesc` (if defined) is the description that will be given in response to an EXAMINE command before the object has been moved (if `initDesc` is not defined, the ordinary `desc` property will be used instead). The definition of `brassCoin` then becomes:

```
brassCoin : Thing '(small) brassy object' 'small brassy object' @longCave
"On the obverse is the head of King Freddie the Fat, and on the reverse
is stamped ONE GROAT. "
initSpecialDesc = "A small brassy object lies on the ground in a dim corner of the cave. "
initDesc = "It looks like it might be a coin of some sort. "
doobjFor (Examine)
{
    action()
    {
        inherited;
        changeName();
    }
}
changeName()
{
    name = 'small brass coin';
    cmdDict.removeWord(self, 'object', &noun);
    initializeVocabWith('brass coin/groat*coins');
}
;
```

Note that `initSpecialDesc` and `initDesc` are only used so long as `moved` is nil; as soon as `moved` is set to true they are no longer employed. The `moved` property is set to nil by the `mainMoveInto(newContainer)`, which is called by `moveIntoForTravel(newContainer)` which is in turn called by `moveInto(newContainer)`, the method most commonly used to move objects in game code or the library's handling of actions like TAKE. Normally this does not matter, but there may be occasions when it could defeat the use of `initSpecialDesc` and `initDesc`. For example, suppose the player had to perform some action to reveal the coin, e.g. because it was hidden under something else or falls out of something else. We might start the coin in another container (or nil) and then move it into the `longCave` using:

```
brassCoin.moveInto(longCave);
```

The trouble is that this will set `brassCoin.moved` to true, so the `initSpecialDesc` and `initExaminedDesc` won't be used, even though this is effectively the first appearance of the coin in the game. The way round this under such circumstances is to set `moved` back to nil in your code:

TADS 3 Tour Guide

```
brassCoin.moveTo(longCave);
brassCoin.moved = nil;
```

4.5. globalParamName

The brassCoin is a little unusual in that it changes its name when it is first examined. This really ought to be reflected in the `initSpecialDesc` property which could instead have been defined as:

```
initSpecialDesc = "\^<<aName>> lies on the ground in a dim corner of the cave. "
```

Then, before the coin is examined it will be listed in a room description as:

"A small brassy object lies on the ground in a dim corner of the cave. "

Whereas if it is examined before being picked up and another LOOK command is issued, it will then appear listed as:

"A small brass coin lies on the ground in a dim corner of the cave."

Which more accurately describes the player's state of knowledge of the object. This is fine, but `globalParamName` allows a slightly neater way of doing the same thing. It's really only useful on objects that change their name in the course of the game (which is likely to be a small minority), and they allow the object to be referred to in a [parameter substitution string](#). This works by setting the `globalParamName` property to a single-quoted string that can be anything we like, but which must be unique (in the realm of parameter names). The `globalParamName` thus set can then be used as a message parameter which refers to this particular object, just as the library parameter `dobj` and `iobj` refer to the direct and indirect objects of the current command. This means we can then rewrite `initSpecialDesc` as

```
initSpecialDesc = "{A coin/he} lies on the ground in a dim corner of the cave. "
```

The definition of the brass coin object then becomes:

```
brassCoin : Thing '(small) brassy object' 'small brassy object' @longCave
  "On the obverse is the head of King Freddie the Fat, and on the reverse
  is stamped ONE GROAT. "
  initSpecialDesc = "{A coin/he} lies on the ground in a dim corner of the cave. "
  initDesc = "It looks like it might be a coin of some sort. "
  globalParamName = 'coin'

  dobjFor (Examine)
  {
    action()
    {
      inherited;
      changeName();
    }
  }
  changeName()
  {
    name = 'small brass coin';
    cmdDict.removeWord(self, 'object', &noun);
    initializeVocabWith('brass coin/groat*coins');
  }
;
```

4.6. specialDesc

There may be cases where we want something other than the plain-vanilla "You see an xxx here" to appear in room description lists even after an object has moved. For this purpose an object may be given a `specialDesc` property as well as an `initSpecialDesc` property. If an object has a `specialDesc` property it is used *either* if the object has moved (i.e. its `moved` property is true) or if there is not also an `initSpecialDesc` property. This works even for objects that

TADS 3 Tour Guide

would not normally be listed, because they are NonPortable. For example, if we wanted the desk in the greatCabin to appear in the list of the cabin's contents we could give it a specialDesc:

```
cabinDesk : Heavy 'large solid oak desk' 'desk' @greatCabin
  "It's a large, solid oak desk. A button is fixed underneath it. "
  specialDesc = "A large oak desks sits in the middle of the cabin. "
;
```

In this case this may a bit redundant, since we have already mentioned the desk in the description of the cabin, and we would normally want one or the other but not both. But at least the specialDesc property allows us the option of which way we do it (although since cabinDesk is never moved it would work equally well to use its initSpecialDesc property). Incidentally, the library does not provide a mechanism for incorporating a specialDesc within the text of a room description (like an Inform describe routine), but it's fairly easy to achieve this effect if you want it, by defining a custom property (say inRoomDesc) on the object you want so described, and a custom method on the room in question, e.g.:

```
greatCabin : Cabin 'Great Cabin' 'the great cabin'
  "The great cabin occupies the entire width of the ship at the stern. The stern
  windows aft look out over the water, while there is a solid wooden bulkhead
  foreward and a flight of steps leads up to the deck above. <<extras>>"
  up = cabinSteps
  fore = bulkheadDoor
  roomParts = static inherited - defaultAftBulkhead - defaultForeBulkhead
    + greatCabinAftBulkhead + greatCabinForeBulkhead
  extras()
  {
    foreach(local cur in contents)
      cur.inRoomDesc;
  }
;

cabinDesk : Heavy 'large solid oak desk' 'desk' @greatCabin
  "It's a large, solid oak desk. A button is fixed underneath it. "
  inRoomDesc = "A large oak desks sits in the middle of the cabin. "
;
```

There is nothing to stop you defining this extras() method on the Room class if you want to make it more general, but you then have to remember to include <<extras>> at the appropriate point of your room descriptions, or else override the library code in some such way as:

```
modify Room
  roomDesc() { inherited; extras; }
  extras()
  {
    foreach(local cur in contents)
      cur.inRoomDesc;
  }
;
```

This may be more convenient, since it will now work in every room without your needing to add <<extras>> to the desc property, provided you're happy for the inRoomDescs always to be listed at the end of the room description. We'll give a more sophisticated version of this modification below.

But to return to specialDesc, we could also use this property to give the coin a more specialized description in a room listing whenever it's dropped on back on the floor, e.g.

```
specialDesc = "{A coin/he} lies on the floor. "
```

Which will give an appropriate description whether the coin has been examined or not. The problem with this is that we want this specialDescription only to be used if the coin is in fact lying on the floor somewhere, and not, for example, if it's placed on some other surface or in some other container. The easiest way to achieve this is to override useSpecialDesc, so that the brassCoin object becomes:

```
brassCoin : Thing '(small) brassy object' 'small brassy object' @longCave
  "On the obverse is the head of King Freddie the Fat, and on the reverse
  is stamped ONE GROAT. "
  initSpecialDesc = "{A coin/he} lies on the ground in a dim corner of the cave. "
  initDesc = "It looks like it might be a coin of some sort. "
  globalParamName = 'coin'
```

TADS 3 Tour Guide

```
specialDesc = "{A coin/he} lies on the floor. "
useSpecialDesc { return location.ofKind(Room) || useInitSpecialDesc(); }
doobjFor (Examine)
{
    action()
    {
        inherited;
        changeName();
    }
}
changeName()
{
    name = 'small brass coin';
    cmdDict.removeWord(self, 'object', &noun);
    initializeVocabWith('brass coin/groat*coins');
}
;
```

You need to be careful about one thing in particular when overriding `useSpecialDesc`, however, namely that `useSpecialDesc` *also* determines if the `initSpecialDesc` is displayed; if `useSpecialDesc` returns `nil` when the `initSpecialDesc` would otherwise be displayed, the `initSpecialDesc` won't be used. The safest way not to fall foul of this problem is to add `|| useInitSpecialDesc()` to whatever condition you're using to determine whether the `specialDesc` should be used, as in the example above (where it is not, in this particular instance, strictly necessary).

To return to our `inRoomDesc` customization, it would be nice if we could choose the order in which objects using our custom `inRoomDesc` property were mentioned in the description of the room that contains them, perhaps by the addition of an `inRoomDescOrder` property. To achieve this, we need to make our customization a bit more complicated:

```
modify Room
roomDesc() { inherited; extras; finalDesc;}
extras()
{
    if(contents.length==0) return;
    local cur;
    local vec = new Vector(10);
    foreach(cur in contents)
        if(cur.propType(&inRoomDesc) is in (TypeDString, TypeCode))
            vec.append(cur);
    if(vec.length==0) return;

    vec = vec.sort(nil, {a, b: a.inRoomDescOrder - b.inRoomDescOrder });
    foreach(cur in vec)
        if(gPlayerChar.canSee(cur))
            cur.inRoomDesc;
}
finalDesc = nil
;

modify Thing
/* Text to add to the description of the room I'm immediately in.
 * If inRoomDesc is a double-quoted string or a method that displays
 * a string, this is added to the description of the enclosing room.
 */
inRoomDesc = nil

/* If several objects in the same room have an inRoomDesc, the inRoomDesc
 * property can be used to define the order in which they are described.
 * To have objects included in the room description in the order in which
 * they are defined in the source file, define inRoomDescOrder = (sourceTextOrder)
 */

inRoomDescOrder = 100
;
```

In the event that you wanted to mix in room description text with object description text in some way other than having all the objects described last, you use the `finalDesc` property, e.g.:

```
boringRoom : Room 'Boring Room'
"There's not much here really, "
```

TADS 3 Tour Guide

```
finalDesc = "The only way out is to the north. "
;

+ Decoration 'carvings' 'carvings'
  "They're rather amateurish. "
  inRoomDesc = "apart from some carvings on one wall. "
;
```

This will produce the room description: "There's not much here really, apart from some carvings on the wall. The only way out is to the north. " This would probably be more useful if the description of the carvings might change, e.g.:

```
+ Decoration 'carvings' 'carvings'
  "They're rather amateurish. "
  inRoomDesc = "apart from some <<epithet>> carvings on one wall. "
  epithet = (described ? 'amateurish' : 'intriguing')
;
```

For a more complex sandwich, you could include SecretFixture objects whose only function was to provide parts of the room description in the sequence determined by their inRoomDescOrder.

4.7. described

The **described** property is simply a flag that indicates whether an object has been explicitly examined by the player. It starts out at nil, and is set to true when the player EXAMINES the object. We can take advantage of this to provide a slightly smoother response if the player first picks up the coin and only then examines it, by explaining on what is then the first examination that the 'small brassy object' is in fact a coin; and while we're at it we can also use it to avoid needlessly calling the changeName() routine more than once (note that this test must come *before* we call the inherited handling, or changeName will never be called):

```
brassCoin : Thing '(small) brassy object' 'small brassy object' @longCave
  "<<described ? nil : 'It turns out to be a coin. '>>
  On the obverse is the head of King Freddie the Fat, and on the reverse
  is stamped ONE GROAT. "
  initSpecialDesc = "{A coin/he} lies on the ground in a dim corner of the cave. "
  initDesc = "It looks like it might be a coin of some sort. "
  globalParamName = 'coin'
  specialDesc = "{A coin/he} lies on the floor. "
  useSpecialDesc { return location.ofKind(Room) || useInitSpecialDesc(); }
  dobjFor(Examine)
  {
    action()
    {
      if (!described) changeName();
      inherited;
    }
  }
  changeName()
  {
    name = 'small brass coin';
    cmdDict.removeWord(self, 'object', &noun);
    initializeVocabWith('brass coin/groat*coins');
  }
;
```

If this is beginning to seem like a lot of complicated work for one simple coin, don't worry; in practice most object definitions are not nearly this elaborate, we have made this one so mainly to illustrate what *can* be done with some of the methods and properties of Thing, not what *must* be done on each occasion. Our small brass coin is now well and truly defined enough, and we shall move on to define some other Things to populate our game world.

4.8. bulk and weight

The **bulk** and **weight** properties are fairly self-explanatory, in that they can be used to hold numbers (which must be integers) representing the bulk (volume) and weight of the item according to any scheme the game author finds convenient.

One use of these properties, which is normally deprecated in modern IF, is to limit what the player character can carry, either by weight or volume. This can be done by setting the player character's `bulkCapacity` and `weightCapacity` properties to some value lower than the default of 10000. Conversely, if you are going to use a large range of numbers for the bulk property of your objects, you might want to raise its `maxSingleBulk` property to something larger than its default value of 10. Although inventory puzzles are now unpopular, it is more acceptable to limit what a PC can carry round in his or her hands if you provide something (such as a bag or sack) he or she can use to transport objects that exceed the capacity of his or her hands.

Another use for the bulk property might be as a rough and ready way of preventing the absurdity of allowing an obviously small container like a purse contain one or more obviously large objects like a packing case or a pair of oars; for this reason alone you might want to give at least a little thought to the bulk you give your objects and the `bulkCapacity` you give any container objects. At the very least it would be odd to have a container whose `bulkCapacity` exceeded its bulk.

Apart from limiting what a player can carry, weight could be used to limit what various platforms and passages can support; you could, for example, have a flimsy bridge that collapses if the total weight it is made to bear exceeds a certain amount. In this game, however, we shall use weight for a different kind of puzzle, namely one that involves putting exactly the correct total weight (which in this game will be 54) on a stone altar in order to open a secret door behind it. Any combination of objects that weigh 54 in total will trigger the secret door, and in due course we shall provide a weighing machine for the player to find out what any portable object weighs. But to make sure the problem is soluble, we shall also provide a set of objects that weigh 1, 2, 4, 8, 16 and 32 units, which guarantees that (once all these objects have been collected) any weight up to 63 units can be formed by some combination of these objects (to obtain 54 the player will need $32 + 16 + 4 + 2$). Each of these objects will be a square tablet, each made of some different material. One face of each of these objects will contain a grid of 25 (5 x 5) letters; when the complete set is collected these inscriptions will, when deciphered, contain the instruction to place 54 pounds on the altar. An inscription on a tombstone outside the temple (in which the altar is located) will provide a clue how the inscriptions are to be deciphered.

Since there will be several of these tablets in the game, all with similar descriptions, it will be convenient to define a `Tablet` class:

```
class Tablet : Thing
    desc = "\^<<theName>> is about eight inches square and an inch thick.
        On it is inscribed:\b<FONT FACE='TADS-Typewriter'><<inscription>></FONT>\b"
    bulk = 4
;
```

We can then define our first tablet and place it in `longCave`:

```
brassTablet : Tablet 'brass tablet*tablets' 'brass tablet' @longCave
    inscription = "F T M T R\nA O O I U\nS T U N L\nT I L R E\nR A D A R"
    initSpecialDesc = "A brass tablet rests by the ladder. "
    weight = 4
;
```

We'll explain how the coded message works later; in the meantime you're welcome to try to work it out for yourself!

4.9. setSuperclassList

It's conceivable that we could have an object that starts out as one kind of thing, but later becomes another. For example we might have a component of something that later proves to be detachable. For example, suppose that the wooden panel becomes detached from the deck rail when it is struck with a heavy hammer. It might seem that this would be impossible to implement since once we have defined something as a `Component`, it is fated to remain a `Component` for the duration. But in fact this is not the case, since in TADS 3 it is possible to change the class list of an object at run-time, using the method **`setSuperclassList(new class list)`**. For example, to make the large wooden panel come free when struck by the hammer we could write:

TADS 3 Tour Guide

```
+ Component 'large wooden panel' 'panel'
...
doObjFor(AttackWith)
{
    verify()
    {
        if(getSuperclassList() != [Component])
            illogicalAlready('You've done it enough damage already! ');
    }

    action()
    {
        if(gIobj == heavyHammer)
        {
            setSuperclassList([Thing]);
            moveInto(getOutermostRoom);
            "The hammer strikes the panel with such force that the panel comes free of the
            rail and falls to the deck. ";
        }
        else
            "{The iobj/he} simply bounces off the panel. ";
    }
}
;
```

Note the use of **getSuperclassList()** to test what class or classes the panel currently belongs to.

The **setSuperclassList** method can be very useful in cases such as this, where the alternative of determining the behaviour of the temporary Component (or Fixture or whatever it may be) by testing the value of some flag in lots of different places would be tedious, long-winded and error-prone. Nevertheless, it's a technique you'll probably want to use sparingly, and with care (we shan't actually be using it at all in *The Quest of the Golden Banana* - the immediately preceding code is hypothetical rather than something to be added to the game). The effect of **setSuperclassList()** is that any methods or properties inherited by the object in question are now inherited from the new list of superclasses, but that properties and methods defined on the object itself remain unaffected (unless they explicitly inherit behaviour). A corresponding transformation will be wrought on anything that inherits from an object (or class) on which **setSuperclassList** is invoked. Obviously this is a tool that needs to be used with some care; it would probably be foolish and reckless to use it to transform a Flashlight into a Actor, or a ComplexContainer into a Candle for example. On the other hand, it may often be useful to transform, say, a Fixture into a Thing (when something previously fixed becomes portable) or a Thing into a Distant (when a portable object goes out of reach - say because it's a flag and we've just hoisted it to the top of the pole).

4.10. Readable

A Readable, as its name suggest, is an object that can be read. In fact you can READ a Thing - it has precisely the same effect as using EXAMINE on it. The main difference between a Thing and a Readable is that on a Readable you can program different responses to READ and EXAMINE. EXAMINE Readable results in the display of its desc property; but READ Readable results in the display of its readDesc (assuming readDesc is defined, i.e. non-nil, otherwise the desc property is displayed).

The other main difference between a Readable and a Thing is that a Readable is regarded as the more logical target of a READ command, so that other things being equal, the parser will choose a Readable object over other kinds of Thing when disambiguating the direct object of READ (i.e. deciding which object the player meant when the command is ambiguous).

Since the tablets all contain squares of letters, they could reasonably be regarded as Readable. We could therefore redefine the Tablet class as:

```
class Tablet : Readable
    desc = "\^<<theName>> is about eight inches square and an inch thick. <<readDesc>>"
    readDesc = "On it is inscribed:\b<FONT FACE='TADS-Typewriter'><<inscription>></FONT>\b"
    bulk = 4
;
```

4.11. Food

The Food class, as its name suggests, is used for things that can be eaten. By default, when eaten, an object of class Food simply disappears (with a default message telling the player that he or she has eaten it). Since food can be eaten it can also be tasted, or smelt. For that matter, it can be touched or felt. To describe what happens when we TASTE it, SMELL it or FEEL it we can use its **tasteDesc**, **smellDesc** and **feelDesc** properties. If you really want to you can even define **soundDesc** to define a response to a LISTEN TO command. Actually, all four of these properties exist on Thing, but this seemed a convenient point at which to introduce them. Later on we shall be looking at more sophisticated ways of handling sensory information. For now we'll just define a banana we'll leave in squareCave:

```
Food 'banana' 'banana' @squareCave
  "It's yellow, about six inches long, and slightly curved. And It looks
  reasonably fresh. "
  tasteDesc = "It's distinctly banana-flavoured. "
  smellDesc = "It has a kind of faint, fruity smell. "
  feelDesc = "The banana skin feels firm but smooth. "
  soundDesc = "The banana is strangely silent. "
  initSpecialDesc = "Someone has left a banana here. "
;
```

4.12. disambigName

If you haven't tried compiling and running the game for a while, now would be a good time to try. Try going into the square cave (using the MEGA or FIAT LUX command to light your path) and then try TASTE BANANA, SMELL BANANA, FEEL BANANA and LISTEN TO BANANA. Then try taking the banana, moving west back into the main cave, dropping the banana, and then trying to take it again with a TAKE BANANA command. At this point you should encounter the following problem:

>take banana

Which banana do you mean, the banana, or the golden banana?

>banana

Which banana do you mean, the banana, or the golden banana?

>

Since we have called our edible banana simply 'banana' there is nothing we can call it that will distinguish it from the golden banana, so in this situation nothing we type will enable us to take the (edible) banana. We could, of course, add edible to its vocabWords, but that won't be apparent to the player, and actually calling it 'edible banana' in its name property would look a bit clumsy. In a case like this the solution is to give it a **disambigName** property, a name that will be used solely for the purpose of disambiguation. We might amend our banana thus:

```
Food '(edible) banana' 'banana' @squareCave
  "It's yellow, about six inches long, and slightly curved. And It looks
  reasonably fresh. "
  tasteDesc = "It's distinctly banana-flavoured. "
  smellDesc = "It has a kind of faint, fruity smell. "
  feelDesc = "The banana skin feels firm but smooth. "
  soundDesc = "The banana is strangely silent. "
  disambigName = 'edible banana'
  initSpecialDesc = "Someone has left a banana here. "
;
```

If you now compile and run the game again, you'll see how using disambigName (coupled with adding 'edible' to the banana's vocabulary) has solved the problem.

4.13. Wearable

A Wearable is simply something that can be worn by an actor. Try defining the following:

```
cap : Wearable 'sailor\'s cap' 'sailor\'s cap' @mainCave
    "It's a large round hat with a white top and a small blue peak. "
;
```

Now recompile the game, go to the mainCave, and try WEAR CAP, INVENTORY, REMOVE CAP, INVENTORY (four separate commands).

The most interesting methods and properties that Wearable introduces are wornBy, isWorn() and isWornBy(actor). None of these are properties or methods you'd normally want to override, but you might have occasion to test their values. **wornBy** returns the actor object that is currently wearing the Wearable (or nil if it is not being worn), **isWorn()** returns true if the Wearable is being worn and nil otherwise, and **isWornBy(actor)** similarly tests for its being worn by a specific Actor. We shall make use of isWorn() shortly, when we add some complications to this cap. Also, we shan't be leaving this cap in mainCave, but it'll have to stay there till we create a new location for it.

5. Containers

5.1. Containers - Introduction

For the purposes of our guided tour of the TADS 3 library, "containers" include every type of physical object that can physically contain another in some way, not only in the obvious sense that the contained object is inside the container, but also where it is on, under or behind the container.

Another way of defining containers in the TADS 3 library is as descendants of the BulkLimiter class:

BulkLimiter

```

BasicContainer
  Container
    Booth
    Dispenser
      Matchbook
    OpenableContainer
      KeyedContainer
      LockableContainer
    RestrictedContainer
    SingleContainer
    StretchyContainer
  SpaceOverlay
    RearContainer
      RearSurface
    Underside
  Surface
    Bed
    Chair
    Platform
      NominalPlatform

```

Some of these will be left to later chapters, since they inherit from other classes we haven't dealt with yet (e.g. Bed, Chair and Platform are all types of NestedRoom, which we'll deal with later, and we'll need to delay discussion of KeyedContainer until we discuss locks and keys in the next chapter). In the present chapter we'll cover the simpler kind of containers. We'll also be covering the following functionally related classes:

```

ComplexComponent
ComplexContainer

```

5.2. BulkLimiter

BulkLimiter is the common base class for containers and surfaces: things that have limited bulk capacities. You probably won't have cause to use this class directly; you'll usually use subclasses such as Surface and Container instead.

BulkLimiter defines the following properties that are inherited by its subclasses:

- **bulkCapacity** - the total aggregate bulk that can be contained in this object (by default, 10000).
- **maxSingleBulk** - the maximum bulk that any individual item inserted into the BulkLimiter may have (by default 10).
- **revealHiddenItems** - a flag that determines whether any Hidden items will be revealed when this BulkLimiter's interior is examined (i.e. when look in, under, or behind will cause the discover method of any item of class Hidden to be called). By default this is true, representing the fact that when we look in, under or behind something we normally see what was there even if we didn't before we looked; if desired this can be set to nil so that Hidden items remain hidden.
- **tooFullMsg** - The message that is displayed when adding a new object would exceed the BulkLimiter's bulkCapacity. This may be overridden on subclasses.
- **becomingTooFullMsg** - the message property to use when doing something to one of our contents would cause our overall contents to exceed our capacity.

TADS 3 Tour Guide

- **becomingTooLargeMsg** - the message property to use when doing something to one of our contents would make it too large to fit all by itself into this container (that is, it would cause that object's bulk to exceed our `maxSingleBulk`).

BulkLimiter also overrides the `notifyInsert()` method to check whether an object will fit into BulkContainer (which it won't if either the aggregate `bulkCapacity` or the individual `maxSingleBulk` would be exceeded by the insertion).

5.3. Surface

Perhaps the simplest kind of container, or BulkLimiter, is the Surface, which is simply something you can put things on. The description of `entranceCave` mentions a narrow ledge carved into one wall, and this would be a good candidate for a Surface; in this case the Surface will also be a Fixture since it's plainly not something we can carry around:

```
Surface, Fixture 'narrow ledge' 'narrow ledge' @entranceCave
    "It's about a foot wide and two feet long. "
    bulkCapacity = 25
;
```

Setting the bulk capacity to 25 isn't essential here, but since the ledge is described as narrow, there must presumably be some limit to how much can be placed on it. If you like you can try running the game and putting things on the ledge.

Another good candidate for a Surface is the desk in the cabin, which is plainly something one could put things on. While we're at it, we'll put something on it:

```
cabinDesk : Heavy, Surface 'large solid oak desk' 'desk' @greatCabin
    "It's a large, solid oak desk. A button is fixed underneath it. "
    inRoomDesc = "A large oak desks sits in the middle of the cabin. "
;

+ chart : Readable 'chart' 'chart'
    "It appears to be a chart of the lake. "
    readDesc = "According to the chart the lake is roughly circular. There appears to
        be one landing spot each on the north, south, east and west shores of the lake. "
    initSpecialDesc = "A chart lies open on the desk. "
;
```

Note the use of the + location here; anything located *in* a Surface is considered to be *on* it. Technically this should cause a problem for our previously defined `Button` object (used to unlock the hidden door in the bulkhead), but the way we've described the desk and the button, together with the fact that the button is a `Component` means that we can in fact get away with it, although later we'll look at a way of tying up this potential loose end.

5.4. BasicContainer

Next to a Surface, the simplest kind of BulkLimiter is a Container, which, as you'd expect, is simply something that can contain other things. The main difference between a Surface and a Container is that whereas the contents of a surface are regarded as being *on* the surface, the contents of a Container are regarded as being *in* the Container.

The other main difference between a Container and a Surface is that, unlike a Surface, a Container can be either **open** or **closed**. If a Container is open its contents are visible and can be removed from the Container, while other things can be inserted into the Container (subject to restrictions of bulk and so forth). If, on the other hand a Container is closed, nothing can be inserted into or removed from it, and, unless the Container is made of some transparent material, its contents will be invisible.

A basic container is an object that can enclose its contents. This is the core of the Container type, but this class only has the bare-bones sense-related enclosing features, without any action implementation. This can be used for cases where an object isn't meant to have its contents be manipulable by the player (so we don't want to allow "put in" and so on), but where we do want the ability to conceal our contents when we're closed.

BasicContainer defines a few properties of its own, of which the most significant are:

TADS 3 Tour Guide

- **isOpen** - defines whether this BasicContainer is open or closed. By default, this property is true. An open box, for example, would have isOpen true, whereas it would be nil on a sealed glass tube.
- **material** - the material from which this container is made; this basically defines whether and how an object in the container can be sensed if the container is closed. The default is *adventium*, which prevents an object in a closed container being sensed at all. If the material were *glass*, we could see what was inside, but not otherwise interact with it. If it were *paper*, we could smell or hear an object in the closed container (assuming it was noisy and smelly) but not see or touch it.

In practice, it's hard to think of examples where this class would be useful (as opposed to one of its subclasses). One possible use would be to have an object permanently encased in a glass container - but then there would seem to be no reason not to have a single object which described itself as a glass container encasing a dead butterfly or whatever it is. On the other hand, if the container can be broken open at some point and the contents removed, never to be replaced, one could use a BasicContainer for that.

To illustrate the fact that if a closed container is transparent you can see its contents but not touch them (and hence not manipulate them), let's create a sealed transparent container with something inside. To make the jar transparent we override its **material** property to **glass**.

```
glassJar : BasicContainer 'glass jar' 'glass jar' @mainCave
  "It seems to be sealed fast. "
  isOpen = nil
  bulkCapacity = 4
  material = glass
;

+ hexCrystal : Thing 'hexagonal blue crystal' 'blue crystal'
  "The crystal is almost cylindrical, except that it has a hexagonal
  cross-section. It's about eight inches long and pulsates with
  a faint blue light. "
  brightness = 1
  bulk = 2
  weight = 2
;
```

Note that since we have described the crystal as pulsating with a faint blue light we give it a brightness of 1 - enough to make it self-illuminating in the dark but not enough for it to illuminate anything else. To see the effect, try carrying the crystal (by carrying the jar) into a dark room. We'll implement a way of [getting the crystal out of the jar](#) shortly.

5.5. Container

Although the plain Container class contains no handling for dealing with OPEN and CLOSE commands from the player (for that you need [OpenableContainer](#) or one of its subclasses), it does have an isOpen property that can be set and manipulated by the author in game code, and, unlike BasicContainer, a Container does allow things to be put inside it in response to a PUT IN command.

One item we have already defined that could be used as a Container, though not obviously so, is the [sailor's cap](#). It won't have a huge capacity, but a cap ought to be able to contain a few small items. Also, it will have the interesting property that it will be closed when worn and open otherwise.

```
cap : Wearable, Container 'sailor\'s cap' 'sailor\'s cap' @mainCave
  "It's a large round hat with a white top and a small blue peak. "
  bulkCapacity = 3
  isOpen { return !isWorn(); }
;
```

Try compiling and running the game, then move to mainCave and experiment with using the cap as a Container when it is and isn't worn (for now you can use the boulder as the object to put in it, though this isn't very realistic). Everything should work fine until you try to put the boulder in the cap while the player character is wearing the cap, whereupon you'll get:

>put boulder in cap

You can't move that through the sailor's cap.

TADS 3 Tour Guide

Although far from disastrous, this is certainly less than ideal. Although you could override the message, a neater solution is to add `objNotWorn` to the preconditions for putting anything in the cap:

```
cap : Wearable, Container 'sailor\'s cap' 'sailor\'s cap' @mainCave
    "It's a large round hat with a white top and a small blue peak. "
    bulkCapacity = 3
    isOpen { return !isWorn(); }
    iobjFor(PutIn) { preCond = static inherited + objNotWorn }
;
```

Then, when the player attempts to put something in the cap while it is worn, a REMOVE CAP command is carried out as an implicit action and the PUT IN command follows (try it and see).

5.6. OpenableContainer

If, unlike the cap and the glass jar in the last two sections, you want a container that can be opened and closed by the player, then you need to use `OpenableContainer` (or one of its lockable subclasses, which we'll be encountering later). As an example of a simple `OpenableContainer` we'll leave a first aid kit on the ledge in `entranceCave` and put a couple of items in it. For convenience, the definition of the ledge is repeated to show the nesting relationship using the + syntax:

```
Surface, Fixture 'narrow ledge' 'narrow ledge' @entranceCave
    "It's about a foot wide and two feet long. "
    bulkCapacity = 25
;

+ firstAidKit : OpenableContainer 'small white first aid box/kit' 'first aid kit'
    "It's made of some kind of white plastic and is about nine inches long. The lid
    is marked with a broad red cross. "
    initSpecialDesc = "A small white box lies on the ledge. "
    bulkCapacity = 3
    bulk = 4
;

++ syringe : Thing 'syringe' 'syringe';

++ stickingPlaster : Thing 'sticking adhesive plaster' 'sticking plaster';
```

Note the use of the + notation to place the `firstAidKit` on the ledge, and the ++ notation to indicate a second level of nesting to put objects in the `firstAidKit`. The use of `initSpecialDesc` means that it will be described as a 'small white box' when the player first encounters it, but will be listed as 'a first aid kit' once the player picks it up, which seems reasonable: its vocabulary has been defined so that it will answer to either appellation. Since it is only a small box we give it quite a small `bulkCapacity`, and a bulk that's just a bit bigger than its capacity. We also place a couple of items in it, but their definition is minimal for now - we'll be fleshing them out in due course.

5.7. notifyInsert & notifyRemove

We are now in a position to implement the scales we can use for weighing the various objects in the game (and so ultimately solve the altar problem that is yet to come). Scales obviously register a new reading each time something is put on them or removed from them, and the best way to test for such occurrences in the TADS 3 library is by using the **`notifyInsert(obj, newCont)`** and **`notifyRemove(obj)`** methods. These have the advantage that they'll also respond to things being inserted into or removed from contents of contents and so forth. In the case of the scales, this means that if I place a box on the scales and then put things in the box or take them out again, the scales' `notifyInsert` and `notifyRemove` methods will still be called, so a change in the total weight on the scales will still be registered, which is what we want.

To get at the total weight on the scales we can simply use the **`getWeight`** method. This returns the total weight of an object and all its contents, so we need to subtract the scales own weight to get the total weight of all the objects placed on it. Since the description of the scales states that the maximum weight it can register is 100 pounds, we need to ensure that it never registers more, however much is placed on the scales. To get at the reading shown by the scales we should thus define:

TADS 3 Tour Guide

```
reading = min((getWeight - weight), 100)
```

There is one major complication, however, and that is that `notifyInsert` and `notifyRemove` are called *before* the insert or remove action is completed, so that at the time they are called, the `reading` property will register the weight on the scales *before* the change, not *after* it as we want. There are probably several ways round this, but the one we have adopted here is to use the **`afterAction()`** method. This is called on all objects (but not rooms) in scope after an action is completed. To achieve the result we want here, we get `afterAction` to test whether the weight on the scales has changed, and only if it has to display the new weight (and record it as the current weight).

The (somewhat complicated) definition of our set of scales is thus:

```
scales : Surface 'large weighing scales/pan/dial/needle' 'scales' @entranceCave
    "These scales comprise a large weighing pan fixed over a square body, on which
    is a large dial with a needle that is currently pointing to <<reading>>. The
    numbers round the dial range from 0 to 100, and according to the inscription
    on the dials the unit of measure is pounds. "
    reading = min((getWeight - weight), 100)
    weight = 6
    isPlural = true
    bulk = 10
    bulkCapacity = 50
    iobjFor(PutIn) asIobjFor(PutOn)
    notifyRemove (obj)
    {
        weighMsg = 'As you remove ' + obj.theName;
    }
    notifyInsert(obj, newCont)
    {
        inherited(obj, newCont);
        weighMsg = 'As you put ' + obj.theName + ' ' + newCont.putInName();
    }
    showWeight()
    {
        "<<weighMsg>> the needle on the dial swings round to <<reading>>. ";
    }
    afterAction()
    {
        if (reading != oldWeight)
        {
            showWeight();
            oldWeight = reading;
        }
    }
    oldWeight = 0
    weighMsg = nil
;
```

There are a number of other points to note here. The first is the use of the **`isPlural`** property. Although the set of scales is in fact a single object, its name property is 'scales', which is grammatically plural; we therefore set `isPlural` to true to ensure that in any message the parser generates about this object the verb will agree in number with its grammatical subject (e.g. to ensure we don't get "The scales does not appear to be edible" when what we want is "The scales do not appear to be edible"). The second is that in this case we can reasonably make the `bulkCapacity` bigger than the `bulk`; there's no reason why a object placed on the scales should not be bigger than the scales. The third is that since the scales are defined as having a pan the player might reasonably PUT X IN PAN as well as PUT X ON SCALES; to handle that we use `iobjFor(PutIn)` as `lobjFor(PutOn)` to translate a PUT IN command to a PUT ON command. The `notifyInsert()` method is already defined on `BulkLimiter`; it already contains code (which, among other things, checks that the object can be inserted and aborts the action if, for example, it is too bulky), so we must call the inherited method. We use the two `notifyXXX` methods simply to start constructing a string that will be displayed if the weight on the scales changes. The `notifyInsert` method also makes use of the **`putInName`** property which returns something like 'in the container' or 'on the surface' as appropriate. Finally, we ensure that the `afterAction()` method only does anything if the weight on the scales has actually changed. Note again that `afterAction` is called after *any* action performed while the object is in scope - this ensures that only actions that change the weight on the scales are acted upon here.

At this point it will be worthwhile to recompile and run the game to test the scales out. Try putting the first-aid box on the scales, then open the first aid box and take the bandage; then try PUT SYRINGE ON SCALES; finally, obtain the brass tablet and try putting it first in the first-aid box and then in the pan. Hopefully, everything should work as expected.

TADS 3 Tour Guide

One small task remains, and that is to put the scales in a plausible locations; we'll place them in a cupboard in a galley aboard the ship, which means we first need to create the galley and the cupboard:

```
galley : DarkCabin 'Galley' 'the galley'
    "It looks like the galley has been more or less stripped bare. There's a work
    surface with a cupboard underneath it, and not much else. "
    aft = crewQuarters
;

+ Surface, Fixture 'work surface' 'work surface';

+ galleyCupboard : OpenableContainer, Fixture '(galley) cupboard' 'cupboard';
```

Then change the first line of the definition of the scales to read:

```
scales : Surface 'large weighing scales/pan/dial/needle' 'scales' @galleyCupboard
```

Clearly, the definition of crewQuarters needs to be changed to reflect this new state of affairs, but we'll attend to that in the next section.

5.8. LockableContainer

A LockableContainer is simply an OpenableContainer that can also be locked and unlocked. This is not as useful as it might sound since a LockableContainer can be locked and unlocked simply by the player issuing LOCK LOCKER and UNLOCK LOCKER commands. Moreover, even if a LockableContainer starts locked, an attempt to OPEN it will result in an implicit UNLOCK command, so that in practice, a LockableContainer behaves much like an OpenableContainer. If you want a container that's locked and unlocked with a key you need to use KeyedContainer, which we'll come to presently.

A simple example of LockableContainer might be locker, which we'll put in the crew quarters:

```
locker : LockableContainer, Fixture '(crew) locker' 'locker' @crewQuarters
    "The locker is fixed firmly to the bulkhead. "
    bulkCapacity = 15
    disambigName = 'crew locker'
    initiallyLocked = true
;
```

Note that if we want a LockableContainer to start locked, we need to set its **initiallyLocked** property to true. The library does this for [Door](#) and IndirectLockable, but you need to do it for anything else (except subclasses of Door, of course).

The sailor's cap would be a good thing to put in the locker, so let's amend its starting location:

```
cap : Wearable, Container 'sailor\'s cap' 'sailor\'s cap' @locker
    "It's a large round hat with a white top and a small blue peak. "
    bulkCapacity = 3
    isOpen { return !isWorn(); }
    iobjFor(PutIn) { preCond = static inherited + objNotWorn }
;
```

At this point we should update the definition of crewQuarters to reflect the presence of the locker and the galley further forward:

```
crewQuarters : DarkCabin 'Crew Quarters' 'the crew quarters'
    "The crew quarters seem largely deserted, apart from a single locker
    fixed to the bulkhead. There's an exit back aft and a ladder leading down into
    the hold. Another exit leads foreward. "
    down = holdLadderDown
    aft = greatCabin
    fore = galley
    cannotGoThatWayInDark()
    ...
;
```

TADS 3 Tour Guide

To make the lock on the locker a bit more worthwhile, we'll suppose that it's fastened by a latch that's rusted shut, and which will only open once we have poured some oil on it. To do this we add a custom oiled property, which we use in the **makeLocked** method. This method is called in response both to a LOCK and an UNLOCK command; its *stat* parameter is true if we want to lock something and nil if we want to unlock it. We can use this method to abort any attempt to lock or unlock the locker until the latch has been oiled. Finally, we add some handling for the PourOnto command on the latch, so that if this latch is the indirect object of PourOnto and the direct object is the oilcan, the oiled property is set to true (which will then allow the locker to be unlocked and opened). Since the player may also try to PULL or PUSH the latch, we add handling for that, making the two commands equivalent. We also redirect any attempts to OPEN, CLOSE, LOCK or UNLOCK the latch back to the locker object.

```
locker : LockableContainer, Fixture '(crew) locker' 'locker' @crewQuarters
  "The locker is fixed firmly to the bulkhead. Its door is fastened by a simple
    latch mechanism, though the latch looks a bit rusty. "
  bulkCapacity = 15
  disambigName = 'crew locker'
  initiallyLocked = true
  makeLocked(stat)
  {
    if(!lockerLatch.oiled)
    {
      reportFailure('The latch is stuck fast. ');
      exit;
    }
    inherited(stat);
  }
;

NameAsOther, SecretFixture targetObj = locker location = crewQuarters;

+ lockerLatch : Component '(locker) latch' 'latch'
  "The latch looks a bit rusty. It's currently in the <<locker.isLocked
    ? nil : 'un' >>locked position. "
  iobjFor(PourOnto)
  {
    verify() { }
    action()
    {
      if(gDobj == oilCan)
      {
        "You pour some oil onto the latch. ";
        oiled = true;
      }
      else
        "It doesn't seem to do much. ";
    }
  }
  dobjFor(Push) asDobjFor(Pull)
  dobjFor(Pull)
  {
    verify() {}
    action()
    {
      locker.makeLocked(!locker.isLocked);
      "This <<isLocked ? nil : 'un'>>locks the locker. ";
    }
  }
  oiled = nil
  disambigName = 'locker latch'
  dobjFor(Open) remapTo(Open, locker)
  dobjFor(Close) remapTo(Close, locker)
  dobjFor(Lock) remapTo(Lock, locker)
  dobjFor(Unlock) remapTo(Unlock, locker)
;
```

A fatally easy mistake to have made here would have been to have made the latch a Component of the locker object. The problem with this would have been that this would have placed the latch *inside* the locker, and therefore not available until the locker was opened (and it's impossible to open the locker without access to the latch, so we'd be in a pretty fix!). For that reason we define another object for the latch to be a Component of (a better way would have been to make the locker a [ComplexContainer](#), but we haven't come to those yet). The player will never interact with this object directly, so it needs no vocabulary. We want it to appear to be the locker when, for example, the player

TADS 3 Tour Guide

attempts to TAKE THE LATCH, so we make it a **NameAsOther** (a mix-in class) and set its **targetObj** property to the latch; the effect of this is that any parser messages referring to this object will describe it in exactly the same way as the latch. We also make the object a **SecretFixture**, since it is an object we need for internal implementation, but not one the player will ever interact with directly.

Note that on the locker we use `exit` to abort the UNLOCK command if `latch.oiled` is nil, and the `reportFailure` macro to explain why the unlock command has failed. The latter is important since the UNLOCK might be an implicit action when the player tries to OPEN the locker; using `reportFailure` here ensures that the implicit action report the player sees then says "(first trying to unlock the locker)" rather than "(first unlocking the locker)". The `PourOnto` handling is fairly straightforward: it tests whether the direct object (`gDobj`) is the oil can, and if so displays an appropriate message and sets the `oiled` property to true, otherwise it displays a non-committal message about not much happening.

We also need to define the oil can. Here we'll provide the minimal definition to do the job. We'll elaborate it later when we use the oil for other purposes (such as fuel for a lamp).

```
oilCan : Thing 'oil can/oilcan' 'can of oil' @secretPassage
  "It's a can full of oil. "
  initSpecialDesc = "An old oil can lies abandoned on the ground. "
  dobjFor(PourOnto) { verify() { } }
;
```

5.9. RestrictedContainer

A Restricted Container is a container that will accept only a limited set of items, defined by the game author.

You may recall that we defined a hexagonal hole in the panel fixed to the quarterdeck rail. This is an obvious candidate for a restricted container, since, as you may by now have guessed, it is designed solely for the hexagonal crystal (for some reason known only in IF Heaven, the ship will only sail when the crystal is in its slot). The definition of the hole needs to be put directly after that of the panel, which we therefore repeat for convenience:

```
+ Component 'large wooden panel' 'panel'
  "The panel seems to have something to do with sailing the ship. A wheel and a lever
  are mounted on it, and between them is a hexagonal aperture. "
;

++ hexHole : RestrictedContainer, Component 'hexagonal hole/aperture' 'hexagonal hole'
  validContents = [hexCrystal]
;
```

Note that we specify what can be put in the hole using its **validContents** property, which contains a list (here containing only a single item) of everything that can be validly inserted. In some cases it might be more convenient to override a **RestrictedContainer's canPutIn(obj)** method. For example if we had defined a **Widget** class and were now defining a **widgetBox** that could only take **Widgets**, we might define its `canPutIn` method as

```
canPutIn(obj) { return obj.ofKind(Widget); }
```

The only difficulty we have right now is that the hexagonal crystal is trapped inside a glass jar, so we can't try inserting it in the hole. Let's assume that one way of getting it out is by cutting the jar open with something suitably hard. First, we'll define a couple of potential cutters (which will also figure later in the game for other purposes):

```
diamond : Thing 'sparkling diamond' 'diamond' @pathEnd
  "It looks like the genuine article. "
  iobjFor(CutWith) { verify() { } }
;

diamondRing : Wearable 'diamond ring' 'diamond ring'
  "It's a fine platinum band with a sparkling solitaire diamond. "
  iobjFor(CutWith) { verify() { } }
;
```

Don't worry that we haven't given any location to the `diamondRing`, the reason will become apparent in due course. Now we can amend our definition of the class `jar` to allow it to be cut open:

TADS 3 Tour Guide

```
glassJar : Container 'glass jar' 'glass jar' @mainCave
  "It <<isOpen ? 'has been cut open' : 'seems to be sealed fast'>>. "
  isOpen = nil
  bulkCapacity = 4
  material = glass
  canBeCutBy = [diamond, diamondRing]
  cannotOpenMsg = (isOpen ? 'It\'s already been cut open' :
    '{You/he} can\'t see any way to open it. ')
  notAContainerMsg = iobjMsg(isOpen ? 'Now that it\'s been cut open, it
    won\'t hold anything. ' : 'There\'s no way
    {you/he} can put anything inside the sealed jar. ')
  dobjFor(CutWith)
  {
    verify()
    {
      if(isOpen) illogicalNow('The glass jar has already been cut open.' );
    }
    check()
    {
      if(canBeCutBy.indexOf(gIobj) == nil)
        failCheck('{You/he} can\'t cut it with {that iobj/him}. ');
    }
    action()
    {
      "{You/he} cut{s} open the glass jar. ";
      isOpen = true;
    }
  }
;
```

Note that `canBeCutBy` is not a library property, it is one we have defined ourselves. It makes it easy to add to the list of items that can be used to cut open the glass jar, should we think of any others at later stage. The `failCheck()` method (a method of `Thing`) was introduced in version 3.0.9. Check methods often contain code like this:

```
check()
{
  if(someCondition)
  {
    reportFailure('There\'s some reason why that won\'t work. ');
    exit;
  }
}
```

Where the `reportFailure` macro tells the parser that the proposed action has failed for some reason (though in practice you could use a double-quoted string) and the `exit` macro terminates processing of the command on this object (and so prevents the action routine from being run). Since this coding pattern is so common, in TADS 3.0.9 it can now be shortened to:

```
check()
{
  if(someCondition)
    failCheck('There\'s some reason why that won\'t work. ');
}
```

Which does exactly the same thing. So the check routine on `glassJar` is exactly equivalent to:

```
check()
{
  if(canBeCutBy.indexOf(gIobj) == nil)
  {
    reportFailure('{You/he} can\'t cut it with {that iobj/him}. ');
    exit;
  }
}
```

It's just that using the `failCheck()` method enables you to code this a little more concisely.

Note also that we have made use of the ability introduced in TADS 3.0.6n to override library messages with our own versions (in this case `cannotOpenMsg` and `notAContainerMsg`) to display something more meaningful in this particular

TADS 3 Tour Guide

case. Note also that in the case of `notAContainerMsg` we have used the `iobjMsg()` macro (new in version 3.0.9), because we only want the customized response to be used when the glass jar is used as the *indirect* object of a command. If we didn't do that we'd see something like:

>PUT COIN IN GLASS JAR

There's no way you can put anything inside the glass jar.

>PUT GLASS JAR IN COIN

There's no way you can put anything inside the glass jar.

This is because unless we specify otherwise, our overridden message will be used whenever the object on which it is overridden is involved in the corresponding command (in this case, a PUT IN command), whether as the direct object or the indirect object. To avoid that we could write:

```
notAContainerMsg = (gIobj == self ? 'My custom message' : nil)
```

Since (as of TADS 3.0.9) if a message property returns `nil` this is taken as meaning "use the standard library message". The `iobjMsg` macro simply makes this a bit easier; allowing us to write the above line as:

```
notAContainerMsg = iobjMsg('My custom message')
```

Which compiles to exactly the same code. If we wanted our custom message to work only when the object its defined on is the *direct* object of a command, we'd use the `dobjFor` macro instead; the following two lines are exactly equivalent:

```
notAContainerMsg = dobjMsg('My custom message')
notAContainerMsg = (gDobj == self ? 'My custom message' : nil)
```

Note that there is no need to use this for a message property for a verb that only take a direct object; e.g., if we define a custom `cannotOpenMsg` property there's no need to use the `dobjMsg` macro since an object can never be the indirect object of an OPEN command.

You may wonder how we know what names to use for these properties: one answer is to look in the library source code to see what message properties are used in the `verify()`, `check()` or `action()` methods of the verbs for which you want to customize the responses.

For example, if we look at the definition of `Thing` in the library code, we find the following handling for when a `Thing` is the indirect object of a `PutIn` command:

```
iobjFor(PutIn)
{
    preCond = [touchObj]
    verify()
    {
        /* by default, objects cannot be put in this object */
        illogical(&notAContainerMsg);
    }
}
```

This means that, left to its own devices, a `Thing` will respond to an attempt to put anything inside it with the message defined in the `notAContainerMsg` of the `playerActionMessages` object. If, however, as here, we define our own version of `notAContainerMsg` on either the direct or indirect object involved in the action, our own version will be used in preference (subject to our use of the `iobjMsg` and `dobjMsg` macros or their long-winded equivalents).

You may, however, find it easier to use the TADS 3 Action Messages quick-reference chart, which you can download either from www.tads.org/t3dl/TemplatesQref.zip or from users.ox.ac.uk/~manc0049/TADSGuide/QRefs.zip.

Note that as of TADS 3.0.10 there are also `RestrictedSurface`, `RestrictedUnderside`, `RestrictedRearSurface` and `RestrictedRearContainer` classes which work analogously to `RestrictedContainer` except that they relate respectively to [Surface](#), [Underside](#), [RearSurface](#) and [RearContainer](#). All these `RestrictedWhatever` classes derive from the common `RestrictedHolder` base class which define `validContents` and `canPutIn(obj)` as described for `RestrictedContainer` above.

5.10. Dispenser

Later in the game the player will use a candle to start exploring the dark areas. In theory the candle could burn out before the player succeeds in finding an alternative light source, thus rendering the game unwinnable. It would thus be desirable for the player to have a large supply of candles available. For this we'll create a box that dispenses candles - a `Dispenser` object. In fact the `Dispenser` doesn't handle much apart from restricting what can be put in it, so this may not be a terribly good example, since we'll have to do most of the work in our own code.

The standard `Dispensable` properties we override on this object will be `myItemClass` and `canReturnItem`. We shall shortly create a `RedCandle` class to be the item dispensed from this box, so we set `myItemClass = RedCandle`. If this were a matchbox we could not return matches to it after we had torn them off, but there seems no reason why we should not return candles to the box, so we set `canReturnItem` to true.

We don't want to create a whole lot of red candles that will never be used - the idea is to allow the player to obtain another one if the one he or she is using burns down before an alternative light source is found. We shall therefore create new candles dynamically on demand; we do this in the `notifyRemove` method. However, to avoid creating candles needlessly, we only create a new one if there's less than two left in the box. Again, we don't want the player to be able to go on obtaining candles *ad infinitum* so we set a maximum (in our custom property `maxCandlesToCreate`) and keep a count of the number created (in the custom property `candlesCreated`). Provided it's okay to create another candle, we do so using the dynamic object creation syntax (`new RedCandle`) and move it into the box. The definition of the `candleBox` then looks like this:

```
candleBox : Dispenser 'large green box' 'large green box' @secretPassage
desc()
{
    "The box is ";
    if(contents.length > 10 || candlesCreated < maxCandlesToCreate/2 )
        "full of red candles. " ;
    else if (contents.length < 1 && candlesCreated == maxCandlesToCreate)
        "empty. ";
    else if(candlesCreated < (3 * maxCandlesToCreate)/4)
        "is about half full of red candles. ";
    else
        "is running out of red candles. ";

}
myItemClass = RedCandle
canReturnItem = true
initSpecialDesc = "A large green box sits by the wall. "
notifyRemove(obj)
{
    if(contents.length < 2 && candlesCreated < maxCandlesToCreate)
    {
        local cur = new RedCandle;
        candlesCreated++;
        cur.moveInto(self);
    }
}
candlesCreated = 0
maxCandlesToCreate = 40
weight = (2 + maxCandlesToCreate - candlesCreated)
bulk = 5
dobjFor(LookIn) asDobjFor(Examine)
;
```

The other things we have done to the `candleBox` is to give it a fairly complex description method which gives a suitable but vague description of its contents, and a calculation of its weight based on the number of candles there are left to create (which must notionally still be in the box). To put an initial red candle in the box we need simply to add:

```
+ RedCandle;
```

But then we have to implement the `RedCandle` class:

```
class RedCandle : Dispensable, Candle 'red candle*candles' 'red candle'
    "It's a long red candle. "
```

TADS 3 Tour Guide

```
isEquivalent = true
isListedInContents = (!isIn(myDispenser))
myDispenser = candleBox
;
```

[Candle](#) is a library class that we'll come to presently. What needs to be noted here is that since all the red candles will be identical, we set **isEquivalent** to true on the class definition; this tells the library that all members of the RedCandle class are functionally identical and interchangeable. This allows you to (say) issue a command TAKE A CANDLE or DROP A CANDLE and have the game respond appropriately even when there are several red candles in scope. It also means that if we pick up three candles and issue an INVENTORY command, we'll be told "You are carrying three red candles" rather than "You are carrying a red candle, a red candle and a red candle." Note that it is important to specify the *candles plural in the vocabWords property so we can issue commands like TAKE TWO CANDLES or DROP BOTH CANDLES.

We set the library **myDispenser** property to candleBox; this simply allows the parser to assume that any command other than TAKE or TAKE FROM directed at a candle is more likely to refer to a candle that's already out of the box. We make further use of this property in an overridden **isListedInContents**, which we set to nil for candles still in their original container. This is to prevent the game announcing the exact number of candles in the box, which would be a misleading number (not taking into account the new candles the box was capable of creating) and would clash with the description we have provided in candleBox.desc.

5.11. StretchyContainer

A StretchyContainer is simply a Container that changes bulk according to its contents. An example might be a sack, which would have virtually no bulk when empty, but becomes bulkier the more is put in it. We can leave one in the squareCave, which could be used for carting things around in:

```
sack : StretchyContainer 'coarse brown sack' 'coarse brown sack' @squareCave
  initSpecialDesc = "A coarse brown sack lies crumpled in the corner. "
  bulkCapacity = 30
  minBulk = 1
;
```

Presumably not even a StretchyContainer is infinitely elastic, so we give it a finite **bulkCapacity**. We can also set a **minBulk** which is the bulk of the sack when empty.

Note that if we want to find out how bulky the sack has become at any point in our game code we need to test its **getBulk()** method, not its bulk property (which never changes).

5.12. SpaceOverlay

You are unlikely to use a SpaceOverlay directly (except perhaps to derive your own subclass from it). The main function of the SpaceOverlay class is to provide common functionality for its subclasses: [Underside](#), [RearContainer](#), and [RearSurface](#). It is worth considering the SpaceOverlay before its subclasses, however, in order to be aware of the common behaviour they all inherit.

According to the comments in the library code:

A "space overlay" is a special type of container whose contents are supposed to be adjacent to the container object (i.e., self), but are not truly contained in the usual sense. This is used to model spatial relationships such as UNDER and BEHIND, which aren't directly supported in the normal containment model.

The special feature of a space overlay is that the contents aren't truly attached to the container object, so they don't move with it the way that the contents of an ordinary container do. For example, suppose we have a space overlay representing a bookcase and the space behind it, so that we can hide a painting behind the bookcase: in this case, moving the bookcase should leave the painting where it was, because it was just sitting there in that space. In the real world, of course, the painting was sitting on the floor all along, so moving the bookcase would have no effect on it; but our spatial relationship model isn't quite as good as reality's, so we have to resort to an extra fix-up step. Specifically, when we move a space overlay, we always check to see if its contents need to be relocated to the place where they were really supposed to be all along.

SpaceOverlay defines the following properties that are inherited by its subclasses:

- **abandonLocation** - This is the location where objects located in a SpaceOverlay (Underside, RearContainer or RearSurface) end up when the SpaceOverlay is moved. By default, this will be the immediate container of the SpaceOverlay. For example, if the SpaceOverlay represents the underside or rear of a dressing table, if the dressing table is moved, then we would expect whatever was behind it to stay put in the dressing table's original location. You can override abandonLocation to some other location if that's where objects in the SpaceOverlay should fetch up, or set it to nil if you want objects in the SpaceOverlay to move with the SpaceOverlay (because they're to be considered attached to the underside or rear of the object that's moved). In addition, any object of class Component in a SpaceOverlay will always move with the SpaceOverlay, since a Component is assumed to be attached to its parent object.
- **alwaysListOnMove** - If this property is set to nil (the default), the SpaceOverlay only lists its contents the first time it's moved (on the basis that if you moved, say, a piece of furniture, you would then see what was behind it or underneath it). If alwaysListOnMove is set to true, on the other hand, then the contents of the SpaceOverlay are listed every time it is moved.

Note that a SpaceOverlay will generally be implemented as a Component of a [ComplexContainer](#): in such a case the 'it' that will actually be moved (causing SpaceOverlays such as its Underside or RearSurface) will be the ComplexContainer (though it will of course take its SpaceOverlays with it).

5.13. Underside

An "underside" is a special type of container that describes its contents as being under the object. This is appropriate for objects that have a space underneath, such as a bed or a table.

Usually, an Underside is not much use by itself (since it would be the Underside of something), and one would normally use it as part of a [ComplexContainer](#). It is, however, possible (though more laborious) to link an Underside to another object using remapTo commands. Just to show how it could be done, we'll give the desk in the cabin an Underside by this means, and then hide the button under it, so that the player can only find it by explicitly looking under the desk:

```
cabinDesk : Heavy, Surface 'large solid oak desk' 'desk' @greatCabin
  "It's a large, solid oak desk, with a single drawer. "
  initSpecialDesc = "A large oak desk sits in the middle of the cabin. "
  specialDescOrder = 50
  dobjFor(LookUnder) remapTo(LookUnder, underDesk)
  iobjFor(PutUnder) remapTo(PutUnder, DirectObject, underDesk)
;

+ underDesk : NameAsOther, Underside, Component
  targetObj = cabinDesk
;

++ Hidden, Button, Component 'small brown button' 'small brown button'
  "The small brown button is fixed to the underside of the desk. "
  dobjFor(Push)
  {
    action()
    {
      "There's a sharp <i>click</i>, and a section of the foreward bulkhead slides
      <<bulkheadDoor.isOpen ? 'closed' : 'open'>>. ";
      bulkheadDoor.makeOpen(!bulkheadDoor.isOpen);
    }
  }
  isListedInContents = (discovered)
;
```

Note that this anticipates the use of the [Hidden](#) class, which we'll be looking at in more detail later.

Apart from a number of message properties, the main new property of interest defined on Underside is **allowPutUnder**; if this is set to true, actors (including the player character) may place objects in (i.e. under) this Underside; otherwise they may not. allowPutUnder is true by default.

5.14. RearContainer

A rear container represents the space behind an object. The principal additional property it defines is **allowPutBehind**; if this is true, objects may be placed in the RearContainer with a PUT BEHIND command; if it is nil, they may not.

For the most part, a RearContainer will be most useful as the ComplexComponent of a [ComplexContainer](#), since it is hard to think of something that only has a rear. A RearContainer can, however, quite successfully be used for an object like a painting or a mirror hanging on a wall, for example:

```
mirror : RearContainer 'large gilt-framed gilt framed mirror'
  'mirror' @anotherCave
  "The mirror is about three foot tall by eighteen inches wide. It is
  brightly silvered, so that your reflection in it is no more flattering
  than you would expect. "
  initSpecialDesc = "A large gilt-framed mirror hangs on the wall opposite
  the dressing table. "
  bulk = 8
  weight = 4
  allowPutBehind = nil
;

+ smallHoleInWall : Hidden, Container, Fixture 'small hole' 'small hole'
  "It's just a couple of inches square, and about as deep. "
  specialDesc = "There's a small hole in the wall
  opposite the dressing table. "
  initSpecialDesc = "Behind the mirror is a small hole in the wall. "
  bulkCapacity = 2
;
```

Once again it has been necessary to anticipate the introduction of the [Hidden](#) class, but it's virtually impossible to illustrate the use of a RearContainer (or other SpaceOverlay) without it, so it'll just have to be taken on trust for now. The effect is that the small hole in the wall will be revealed only when the player looks behind the mirror or takes it for the first time. Also, when the mirror is moved, the small hole is moved from the mirror to the mirror's former location, which paradoxically has the effect of leaving it behind in the same place. This occurs even though the small hole is a fixture, so that after the mirror is moved, the hole ends up being a Fixture in the room, which is what we want.

Note that we have set allowPutBehind to nil to prevent anything being put behind the mirror; which would normally make sense (since it would normally not be that easy to put sundry objects behind a mirror hanging on the wall). In this case, however, we might feel that while the mirror is still hanging on the wall, putting something behind the mirror is equivalent to putting it in the hole, but that it should not be possible to put anything behind the mirror once it's been moved. We can implement this like so:

```
mirror : RearContainer 'large gilt-framed gilt framed mirror'
  'mirror' @anotherCave
  "The mirror is about three foot tall by eighteen inches wide. It is
  brightly silvered, so that your reflection in it is no more flattering
  than you would expect. "
  initSpecialDesc = "A large gilt-framed mirror hangs on the wall opposite
  the dressing table. "
  bulk = 8
  weight = 4
  allowPutBehind = (!moved)
  iobjFor(PutBehind) maybeRemapTo(!moved, PutIn, DirectObject, smallHoleInWall)
;
```

In due course, we'll hide a small piece of black wire in the small hole, but we'll wait till we get to the point when this bit of wire is needed and we've covered the ground we need to implement it properly. In the meantime, there's one further detail to attend to. As things are at the moment, when you take the mirror the transcript goes something like this:

>take mirror

Behind the mirror is a small hole in the wall. Taken.

In this case it's reasonably obvious that 'Taken' must refer to the mirror and not the small hole, but it's not as clear as it might be, and in other circumstances, where what lay behind or beneath something was a portable object that easily

TADS 3 Tour Guide

could be taken, the 'Taken' message might be downright misleading. We can make the message clearer by making the following modification to SpaceOverlay:

```
modify SpaceOverlay
  okayTakeMsg = '{You/he} take{s} {the dobj/him} '
;
```

Now, to return to our ship, since the chair in the main cabin is described as being behind the desk, it may be tempting to try this:

```
cabinDesk : Heavy, Surface 'large solid oak desk' 'desk' @greatCabin
  "It's a large, solid oak desk, with a single drawer. "
  initSpecialDesc = "A large oak desk sits in the middle of the cabin. "
  specialDescOrder = 50
  dobjFor(LookUnder) remapTo(LookUnder, underDesk)
  iobjFor(PutUnder) remapTo(PutUnder, DirectObject, underDesk)
  dobjFor(LookBehind) remapTo(LookBehind, deskRear)
  iobjFor(PutBehind) remapTo(PutBehind, DirectObject, deskRear)
;

+ deskRear : NameAsOther, RearContainer, Component
  targetObj = cabinDesk
;

cabinChair : Chair 'padded chair/cushion' 'chair' @deskRear
  "It's a fine wooden chair with a round back and a padded cushion. "
  initSpecialDesc = "A wooden chair sits behind the desk. "
  bulk = 10
  weight = 7
;
```

This appears to work well enough, in that you can look behind the desk and be told that the chair is there, or take the chair, subsequently put it behind the desk and find that it's described as being there once more, but you'll quickly discover that it all goes horribly wrong if you try to sit on the chair while it's behind the desk.

It takes quite a bit of work to fix this, which will involve classes and concepts we haven't met yet (particularly the [Platform](#) class) . But to show what we need to do, here it is:

```
cabinDesk : Heavy, Surface 'large solid oak desk' 'desk' @greatCabin
  "It's a large, solid oak desk, with a single drawer. "
  initSpecialDesc = "A large oak desk sits in the middle of the cabin. "
  specialDescOrder = 50
  iobjFor(PutUnder) remapTo(PutUnder, DirectObject, underDesk)
  dobjFor(LookBehind) remapTo(LookBehind, deskRear)
  iobjFor(PutBehind) remapTo(PutBehind, DirectObject, deskRear)
  dobjFor(GoBehind) remapTo(GoBehind, deskRear)
;

deskRear : RearContainer, Platform, SecretFixture
  name = 'desk'
  actorInPrep = 'behind'
  actorIntoPrep = 'behind'
  actorOutOfPrep = 'from behind'
  location = greatCabin
  dobjFor(GoBehind)
  {
    verify() { logicalRank(140, 'rear'); }
    action()
    {
      gActor.moveIntoForTravel(self);
      defaultReport('{You/he} go{es} behind {the dobj/him} ');
    }
  }
  tryMovingIntoNested()
  {
    return tryImplicitAction(GoBehind, self);
  }
;
```


TADS 3 Tour Guide

```
DefineTAction(GoBehind)
;

VerbRule(GoBehind)
('go' | 'stand' | 'walk') 'behind' singleDobj
: GoBehindAction
verbPhrase = 'go/going (behind what)'
;

modify Thing
dobjFor(GoBehind)
{
    verify() { illogical('{You/he} can\'t go behind {that dobj/him}. '); }
}
;
```

This seems a quite a lot of work to be able to put a chair behind a desk (and even then one or two the messages displayed may be less than ideal); but if you really *want* a chair behind the desk, it may be worthwhile.

5.15. RearSurface

A `RearSurface` is simply a `RearContainer` for which `abandonLocation` is nil by default, meaning that the contents of a `RearSurface` are considered to be attached to it and so move with it. As the comments in the library code explain it:

A "rear surface" is essentially the same as a "rear container," but models the contents as being attached to the back of the object rather than merely sitting behind it.

The only practical difference between the "container" and the "surface" is that moving a surface moves its contents along with it, whereas moving a container abandons the contents, leaving them behind where the container used to be.

As with `RearContainer`, you'd be most likely to use `RearSurface` as the `ComplexComponent` of a `ComplexContainer`, but you could also use it for a flat object that has something attached to back. For example, we could have a small photo with a mysterious piece of paper attached loosely to the back (we'll return to this piece of paper later). For now you can put this pair of objects in any convenient location; we'll be assigning them their proper home later:

```
+ smallPicture : RearSurface 'small picture' 'small picture'
  "It's a faded photograph of an eccentrically-dressed man with a
    long scarf, in company with a smiling young woman with
    long blonde hair. "
  allowPutBehind = nil
;

++ rightHalfPaper : Hidden, Readable 'right half torn sheet yellow paper*sheets'
'torn sheet of yellow paper'
  "It looks like the left hand half of a sheet of paper that's been torn in two. It
    contains a list of names. "
;
```

Once again, we make this piece of paper `Hidden` so the player won't find it without looking behind the picture. The piece of paper will move with the picture, but will readily detach from it when taken, thereby modelling a piece of paper that is only loosely attached.

One final detail we can handle is that it may occur to the player to look at the rear of the picture, but to do so by typing `LOOK AT BACK OF PICTURE` instead of the, perhaps less natural and less obvious, `LOOK BEHIND PICTURE`. This can be handled quite readily by adding the following:

```
++ Decoration 'back/picture/photo/photograph' 'back of the picture'
dobjFor(Examine)
{
    verify() { nonObvious; }
    action() { replaceAction(LookBehind, smallPicture); }
}
;
```

The thing to note here is our use of the `nonObvious` in the `verify` routine; this is to prevent the back of the picture being included by the parser as a likely target of the `EXAMINE` command, so that a command like `X PICTURE` doesn't give the game away by responding with "Which do picture do you mean, the small picture or the back of the picture?"

5.16. ComplexContainer

As we saw with the potential trap in trying to add a `Component` to a `LockableContainer`, an item's contents are deemed to be either in it or on it (or, by extension, under it or behind it), but only one of these at a time. So what happens if we have something like `desk` that we want to put things both on and in? Well, we've already seen how one solution might work, because we've been using it with [Underside](#) and [RearContainer](#), namely to define a second object, say a desk drawer, to act as the container, and remap all the commands relating to looking in, opening, closing, and putting things into the desk to the drawer object instead:

```
cabinDesk : Heavy, Surface 'large solid oak desk' 'desk' @greatCabin
  "It's a large, solid oak desk, with a single drawer. "
  initSpecialDesc = "A large oak desk sits in the middle of the cabin. "
  specialDescOrder = 50
  dobjFor(Open) remapTo(Open, cabinDeskDrawer)
  dobjFor(Close) remapTo(Close, cabinDeskDrawer)
  dobjFor(LookIn) remapTo(LookIn, cabinDeskDrawer)
  iobjFor(PutIn) remapTo(PutIn, DirectObject, cabinDeskDrawer)
  dobjFor(LookUnder) remapTo(LookUnder, underDesk)
  iobjFor(PutUnder) remapTo(PutUnder, DirectObject, underDesk)
  dobjFor(LookBehind) remapTo(LookBehind, deskRear)
  iobjFor(PutBehind) remapTo(PutBehind, DirectObject, deskRear)
  dobjFor(GoBehind) remapTo(GoBehind, deskRear)
;

+ cabinDeskDrawer : OpenableContainer, Component 'drawer' 'drawer'
  bulkCapacity = 4
;

++ tardisKey : Key 'small silver key' 'small silver key';
```

Note the use of `DirectObject` in the `remapTo(PutIn...)` macro when we are remapping a command for which the desk is the indirect object. Again, we have defined the key to be of class `Key` which we haven't actually introduced yet, but since the object definition is so simple it would be pointless to make it a `Thing` only to have to change it later. We'll see how this `Key` works when we come to discuss the [LockableWithKey](#) class.

Note also that although the `cabinDeskDrawer`, being in the `cabinDesk`'s contents, is strictly speaking notionally on its surface, this doesn't matter in practice, since as a `Component` it will never be listed, and since it's on the outside of the desk and not within some kind of `Container` there's no danger of its being inappropriately hidden from the player.

It would be perfectly feasible to implement all objects that have contents both on them and in them in this way. There is, however, another way, or rather a way that automates some of the labour, and that is to use a `ComplexContainer`. We'll put an old dressing-table in anotherCave as an example:

```
dressingTable : ComplexContainer, Heavy 'battered old dressing table' 'dressing table'
  @anotherCave
  "It's battered and scratched, and looks just about on its last legs. In place
  of drawers, it has a pair of doors attached to the front"
  inRoomDesc = "A battered old dressing table leans drunkenly against a wall of the cave. "
  subSurface : ComplexComponent, Surface { }
  subContainer : ComplexComponent, OpenableContainer
  {
    bulkCapacity = 5
    openStatus { if(isOpen) ". It's open"; }
  }
  subUnderside : ComplexComponent, Underside { bulkCapacity = 5 }
  subRear : ComplexComponent, RearContainer { bulkCapacity = 5 }
;
```

The main thing to note here is the way the `subSurface`, `subContainer`, `subUnderside` and `subRear` properties are defined. Each must contain the definition of a nested object of class `ComplexComponent`, together with an appropriate

TADS 3 Tour Guide

Surface class (for subSurface) or Container class (for subContainer) or Underside (for subUnderside) or RearContainer class (for subRear). The ComplexContainer will then direct the relevant commands (e.g. PUT ON, PUT IN, OPEN, CLOSE, LOOK IN, LOOK UNDER, LOOK BEHIND) to its subSurface, subContainer, subUnderside or subRear as appropriate. The definition of the ComplexComponent objects can be as minimal as the subSurface, or we can introduce further customization, as with the subContainer. Here we give the table a limited bulkCapacity when it acts as a Container, and override its **openStatus()** method to suppress the "It's closed. " message that would otherwise be appended to the description of the dressing table in response to an EXAMINE command (we have to cheat a little here, since even if openStatus prints nothing, a terminating full stop will be printed; so we omit the full stop from the end of the desc property and put one at the start of ". It's open"; that way the description will look right whether the table is open or closed).

Note that [inRoomDesc](#) is *not* a property defined in the library; it was a custom property we defined some way back as a convenient way of adding the description of NonPortable objects to the room description. Here this simply avoids our having to go back and change anotherRoom.desc.

The one thing that may not be immediately obvious is how to define the initial location of objects in or on a Complex container. This is one way:

```
silverCoin : Thing 'small silver coin' 'small silver coin'
  "On the obverse is the head of Queen Fanny the Futile; the reverse is stamped with
  the words THREE FARTHINGS. "
  location = dressingTable.subSurface
;

ring : Thing 'platinum ring/band/recess' 'platinum ring'
  "It's a plain platinum band, with a small empty recess on one side. "
  location = dressingTable.subContainer
;
```

And this is the other:

```
+ silverCoin : Thing 'small silver coin' 'small silver coin'
  "On the obverse is the head of Queen Fanny the Futile; the reverse is stamped with
  the words THREE FARTHINGS. "
  subLocation = &subSurface
;

+ ring : Thing 'platinum ring/band/recess' 'platinum ring'
  "It's a plain platinum band, with a small empty recess on one side. "
  subLocation = &subContainer
;
```

Whichever way you use, you should only use the location or subLocation property to set the *initial* location of an item. To move an item into part of a ComplexContainer you should use moveInto(), e.g. ring.moveInto(dressingTable.subRear).

ComplexContainer Traps for the Unwary

Although ComplexContainers can be very useful, they can also be the source of very strange, frustrating and hard-to-trace bugs in your code. The reason is that after you've set your ComplexContainer up, it can be very easy to forget that, programmatically, it is not a single object but a linked collection of objects. Thus, for example, you might later write a routine to check the contents of all the containers held by a particular actor, and do something like:

```
foreach(local cur in actor.contents)
  if(cur.ofKind(Container))
    foreach(local obj in cur.contents)
    {
      /* do something with the contents */
    }
```

The problem here is that you may, for some reason, have implemented a portable container as a ComplexContainer; perhaps it's an openable box with a handle on the lid; to make the handle a Component which doesn't disappear when the box is closed, you have to make the box a ComplexContainer. You find the above code is mysteriously excluding one of the containers, which you finally realize is because it's a ComplexContainer, so you amend the code to:

```
foreach(local cur in actor.contents)
  if(cur.ofKind(Container) || cur.ofKind(ComplexContainer))
```

TADS 3 Tour Guide

```
foreach(local obj in cur.contents)
{
    /* do something with the contents */
}
```

But when this gets to your box, the above code won't work as expected, since the 'contents' of the `ComplexContainer` will be its `subContainers` and its `Component` handle, not the things inside the box, as you automatically expected. The objects actually held inside the box are actually to be found in its `subContainer.contents` property, not its `contents` property. What you actually need in the above example is something like:

```
foreach(local cur in actor.contents)
{
    local cont = [];
    if(cur.ofKind(Container))
        cont = contents;
    if(cur.ofKind(ComplexContainer) && (cur.subContainer != nil))
        cont = cur.subContainer.contents;
    foreach(local obj in cont)
    {
        /* do something with the contents */
    }
}
```

Prior to TADS 3.0.8 there was also a potential trap with opening, closing, locking and unlocking `ComplexContainers`, particularly a `ComplexContainer` that you came to think of as being primarily a container. You might, for example, test for `cupboard.isOpen` when you needed to test for `cupboard.subContainer.isOpen`, or call `cupboard.makeLocked(true)` when you actually meant `cupboard.subContainer.makeLocked(true)`.

Fortunately TADS 3.0.8 introduced some changes that greatly alleviates this. Suppose you have a cupboard mounted on the wall, which you can put things in, on or under. The cupboard will have a `subSurface`, a `subContainer` and a `subUnderside`. If you open, close, lock or unlock the cupboard, you are actually opening, closing, locking or unlocking its `subContainer`. As of version 3.0.8 TADS has `ComplexContainer` recognize this by having its `isOpen`, `isLocked`, `makeOpen` and `makeLocked` properties and methods refer to the corresponding properties and methods of its `subContainer`, provided it has one. So, for example, if you test for `cupboard.isOpen` you will now get the value of `cupboard.subContainer.isOpen`. Likewise, if you write a statement like `cupboard.makeLocked(true)` it will now automatically call `cupboard.subContainer.makeLocked(true)`. Of course you can, if you wish, continue to test explicitly for `cupboard.subContainer.isOpen` and explicitly code `cupboard.subContainer.makeLocked(true)`; the point is not so much that these forms are more long-winded, but that it can be very easy to forget to do this, especially if you come to think of the cupboard as being primarily a form of lockable and openable container (which just happens to allow things to be put on top of it and underneath it as well).

5.17. ContainerDoor

The purpose of a `ContainerDoor` is to represent the door of a `Container`, when the player might want to refer to it separately. A container's door cannot be straightforwardly made a component of its container (located in the container), since this would have the effect of putting the door *inside* the container, where it would vanish out of sight when the container was closed. You would therefore normally use a `ContainerDoor` as a component of (located in) a [ComplexContainer](#); it then maps OPEN, CLOSE, LOCK, UNLOCK, LOOK IN and LOOK BEHIND commands to the `subContainer` of that `ComplexContainer`.

For example, the dressing table we've just defined mentions in its description that it has a pair of doors. To implement those doors we just need to define the following, immediately after the definition of the dressing table `ComplexContainer` object:

```
+ ContainerDoor '(dressing) (table) door/pair/doors' 'dressing table door'
    "They're made of the same scratched, stained wood as the dressing table
    to which they're attached, and perfectly match its generally battered
    appearance. "
    isPlural = true
;
```

It's also possible to set up a `ContainerDoor` to act as the door to any kind of openable container, by setting its **subContainer** property to point to that container. For example, we could add a door to the locker in the crew quarters by this means:

TADS 3 Tour Guide

```
locker : LockableContainer, Fixture '(crew) locker' 'locker' @crewQuarters
    "The locker is fixed firmly to the bulkhead. Its door is fastened by a simple
    latch mechanism, though the latch looks a bit rusty. "
    bulkCapacity = 15
    disambigName = 'crew locker'
    initiallyLocked = true
    makeLocked(stat)
    {
        if(!lockerLatch.oiled)
        {
            reportFailure('The latch is stuck fast. ');
            exit;
        }
        inherited(stat);
    }
;

NameAsOther, SecretFixture targetObj = locker location = crewQuarters;

+ ContainerDoor '(locker) door' 'locker door'
    "The locker door is a plain wooden front, fastened by a latch. "
    subContainer = locker
;

++ lockerLatch : Component '(locker) latch' 'latch'
    "The latch looks a bit rusty. It's currently in the <<locker.isLocked
    ? nil : 'un' >>locked position. "
    ...
;
```

Of course, it would probably have been easier to make the locker a `ComplexContainer` and attach the `ContainerDoor` to that; but at least this shows that other arrangements are possible.

5.18. SingleContainer

A `SingleContainer` is a special type of container that can hold only one object at a time. If a new object is inserted, the old one is removed.

The example we'll create may seem a little contrived at first, but hopefully it'll make more sense when it ends up in its proper context. For now we'll simply put it in `mainCave`, where it'll be convenient to test it out until its proper starting place (a space station we'll be visiting in a *Tardis*) has been constructed. From the description of this object it's fairly clear we'll need to add a number of components to it in due course. Since we don't want these to end up inside in the `Container` we make our `autoRectifier` a `ComplexContainer` and the `SingleContainer` its `subContainer` object:

```
autoRectifier : ComplexContainer 'silver cylinder' 'silver cylinder' @mainCave
    "It's about a foot high and five inches in diameter. A black ring surrounds
    the opening at one end. The only other feature of interest are a conspicuous
    orange button and the manufacturer's name inscribed just below the ring. "
    subContainer : ComplexComponent, SingleContainer { bulkCapacity = 3 }
    bulk = 4
    weight = 3
;
```

To try this out, compile the game, pick up the first aid kit on the way down to the main cave, then try putting the contents of the first aid kit in the cylinder one at a time.

This gadget is clearly incomplete, but we'll add its components and make it functional when we come to deal with the [bent key](#) below.

5.19. BagOfHolding

BagOfHolding is a mix-in class that can be used for an object to which an actor (usually the player character) automatically moves objects when his or her hands become full, provided the BagOfHolding object is in the player's inventory. This provides a measure of realism (there's a limit to how many objects an actor could really hold in his or her hand) without making inventory management too burdensome to the player.

All that's necessary to make an object a BagOfHolding is to add BagOfHolding to the start of its class list; a good candidate for a BagOfHolding in the *Quest of the Golden Banana* might be the sack we defined earlier:

```
sack : BagOfHolding, StretchyContainer 'coarse brown sack' 'coarse brown sack' @squareCave
    initSpecialDesc = "A coarse brown sack lies crumpled in the corner. "
    bulkCapacity = 3000
    minBulk = 1

    affinityFor(obj)
    {
        return obj.ofKind(Tablet) ? 200 : inherited(obj);
    }
;
```

If it's to be much use as a BagOfHolding the sack will need a bulkCapacity much larger than the one we initially gave it, so here we increase its bulkCapacity to 3000.

Although there's no particular reason for doing it in this case, we define the sack's `affinityFor()` method just to illustrate its use. This should return a number between 0 (meaning that the BagOfHolding will refuse to hold the object) to 200 (meaning that the bag is particularly keen to hold the object), with 100 being the default. In this case we'll make the sack particularly suitable for carrying the various tablets in (there's no really *logical* reason for this beyond player convenience - the tablets are relatively bulky objects that the player needs to collect a number of but not to use very often).

Note that we won't see the BagOfHolding doing much unless we also reduce the bulk capacity of the player character. You could try reducing it to 100; add the following to the definition of the me object:

```
bulkCapacity = 100
```

6. Locks & Keys

6.1. Locks & Keys - Introduction

We have already met the LockableContainer class. We shall now go on to look at other types of Lockable objects, including those that use keys. The relevant classes are:

Lockable

IndirectLockable

LockableContainer

LockableWithKey

KeyedContainer

Key

KeyRing

6.2. Lockable

Lockable is a mix-in class that must be used with other classes such as Door or Openable, but even when mixed-in the Lockable class doesn't really do much, as we saw in the case of the LockableContainer, since it simply allows something to be locked and unlocked via LOCK and UNLOCK commands, which are carried out implicitly if the player tries to open whatever it is that's locked.

You can verify this by modifying both sides of the door into and out of the lakeRoom:

```
+ lakeDoor : Lockable, Door 'door' 'door';
...
+ lakeDoor2 : Lockable, Door ->lakeDoor 'door' 'door';
```

If you compile and run the game and try to go south through this door from anotherCave you'll find the lock doesn't prove much of a barrier. The only reason to use plain vanilla Lockable with a Door is if the other side of the door is going to be locked by some less plain vanilla means, which is what we'll go on to do. This could represent a situation like a front door, say, which is locked and unlocked by a key on the outside and a simple knob on the inside. Locking the door on the inside would then prevent pursuit by an Actor who did not have the key.

There are a number of properties and methods on the Lockable class, the most useful of which for game authors are:

- **autoUnlockOnOpen**: if true the object is automatically unlocked when someone attempts to open it (or at least, the parser attempts to unlock it, although the attempt may fail if there's some reason why the object can't be unlocked). The library default is to set this to lockStatusObvious (see below).
- **lockedDesc**: the description to display when the object is locked or unlocked. The library default is fine for most situations, but if you want to customize it you need to define the property in the form lockedDesc = (isLocked() ? messageWhenLocked : messageWhenUnlocked), where messageWhenLocked and messageWhenUnlocked are single quoted strings or properties/methods evaluating to single quoted strings.
- **lockStatusObvious**: this should be true or nil depending on whether an actor should be able to tell by simple visual inspection that the object is locked. The library default is true. For a LockableWithKey (e.g. a door with a keyhole) it might not be obvious whether the object is locked until the player tries to open it. In such a case the most desirable behaviour might be for the game to change lockStatusObvious from nil to true once the door has been tried; for example you could override cannotUnlockMsg on the object to include something like '<.reveal door-locked>' and then set lockStatusObvious = gRevealed('door-locked').
- **lockStatusReportable**: this is used to decide whether the parser should report the object as being locked or unlocked. For example, if an object is open, it is obviously unlocked, so it is redundant to report something like "The door is red. It's open. It's unlocked. ", it is sufficient to report "The door is red. It's open. " The library takes care of this particular case by default, but there may be other cases where you want to override the library behaviour.

TADS 3 Tour Guide

- **isLocked()**: note this is a *method*, not a property; test this value to determine whether the object is locked, but do *not* overwrite it to lock or unlock an object programmatically. Call `makeLocked()` instead.
- **makeLocked(stat)**: Call `makeLocked(true)` or `makeLocked(nil)` to lock or unlock the object programmatically (e.g. as the result of pushing a button or pulling a lever on an [IndirectLockable](#)). You can also override this method if you want to produce some additional side-effects of locking or unlocking the object, but make sure you then call `inherited(stat)` somewhere in your `makeLocked(stat)` method.
- **initiallyLocked**: if this is true (as it is by default) then the Lockable object starts out locked, so if we don't want it to start out locked we need to change this to nil (note that this was first added to Lockable in TADS 3.0.10)

IMPORTANT NOTE. Since Lockable is a mix-in class (not derived from Thing) (1) it cannot be used on its own (you can't define a physical object as being simply Lockable, it must be a Lockable something, such as a Lockable, Container or a Lockable, Door) and (2) it must be listed *before* any Thing-derived class it is mixed-in with. Thus whereas the following works fine:

```
+ lakeDoor : Lockable, Door 'door' 'door';
```

The following does not:

```
+ lakeDoor : Door, Lockable 'door' 'door';
```

The second of these will compile fine, and the door will appear - but the lock won't work as expected (for example, even if the `initiallyLocked` property is set to true, the door won't start out locked).

6.3. IndirectLockable

IndirectLockable is another mix-in class for use with objects such as doors that cannot be unlocked with a simple UNLOCK command, but do not use a key; that is something that must be unlocked by some other mechanism such as a lever or switch. To make things more interesting we'll change the door into the lakeRoom into an indirectLockable, which is unlocked by inserting the brass coin into a slot:

```
+ lakeDoor : IndirectLockable, Door 'smart new door' 'smart new door'
    "The door is completely plain apart from a small vertical slot. "
    cannotUnlockMsg = 'The door does not appear to have a conventional lock. '
;

++ RestrictedContainer, Component 'small vertical slot' 'slot'
    "It's about half an inch long; next to it is some faded writing that
    just about still says\nENTRANCE TO LAKE\nONE GROAT. "
    validContents = [silverCoin, brassCoin]
    notifyInsert(obj, newCont)
    {
        if(obj==brassCoin)
        {
            "As the brass coin disappears into the slot you hear a click from the door. ";
            obj.moveTo(nil);
            lakeDoor.makeLocked(nil);
        }
        else
        {
            "Despite initial appearances <<obj.theName>> doesn't seem to be quite
            right for the slot. ";
        }
        exit;
    }
;
```

We make use of a RestrictedContainer to accept the coin, but since we have left a silver coin in plain sight next to the door the player is almost bound to try it, so we include it in the list of validContents so that we can display a custom message for it. We handle the insertion of either coin in the notifyInsert method. If the coin is the brass one, we display a suitable message, move the coin into nil (since it presumably disappears into some repository) and unlock the lakeDoor by calling its **makeLocked** method: `makeLocked(nil)` unlocks the door, whereas `makeLocked(true)` would lock it again (which is not something we want to do here). Whichever coin was inserted we end notifyInsert with an exit statement since we do not in fact want the rest of the default command handling, which would move the coin into the slot.

TADS 3 Tour Guide

Note also the use of the **cannotUnlockMsg** property to provide a customized response to attempts to unlock the door other than by inserting the coin.

The **initiallyLocked** property of an **IndirectLockable** controls whether it starts life locked (if this property is true) or unlocked (if its nil). By default, **initiallyLocked** is true.

Like **Lockable**, **IndirectLockable** is a mix-in class that *must* precede any Thing-derived class it is mixed in with, as in:

```
+ lakeDoor : IndirectLockable, Door 'smart new door' 'smart new door'
    "The door is completely plain apart from a small vertical slot. "
    cannotUnlockMsg = 'The door does not appear to have a conventional lock. '
;
```

The following would have been wrong (since the door would not have started out locked):

```
+ lakeDoor : Door, IndirectLockable 'smart new door' 'smart new door'
    "The door is completely plain apart from a small vertical slot. "
    cannotUnlockMsg = 'The door does not appear to have a conventional lock. '
;
```

6.4. KeyedContainer

A **KeyedContainer** is a **Container** that can be opened and closed, and also locked and unlocked with a key. As an example we'll put a large, heavy trunk in **mainCave**:

```
trunk : KeyedContainer, Heavy 'large black trunk' 'large black trunk' @mainCave
    initSpecialDesc = "A large black trunk rests in the middle of the cave. "
    initiallyLocked = true
    keyList = [brassKey]
;
```

Note that we have to set **initiallyLocked** to true if we want the trunk to start locked, and that **keyList** needs to be set to the list of keys that can lock and unlock this container. Clearly, we also need to define the key, which is entirely straightforward:

```
brassKey : Key 'small brass key' 'brass key' @roundCave;
```

Note that even if it is listed in another objects' **keyList**, to function as a key an object must be of class **Key** and not simply **Thing**; the **Key** class contains a number of specializations, of which the most important is overriding **verifyobjLockWith()** and **verifyobjUnlockWith()** to make it logical (which implies possible) for a **Key** to be used as the indirect object of these commands.

The trunk will obviously be more interesting if there's something inside when it's opened, so let's put the glass jar and the fake golden banana inside:

```
fakeTreasure : Thing 'huge great golden gold banana/treasure'
    'golden banana' @trunk
...
;

glassJar : BasicContainer 'glass jar' 'glass jar' @trunk
...
;
```

You'll also need to delete **fakeTreasure**'s **initSpecialDesc**, which is no longer appropriate.

Finding a key and opening a container with it is a pretty standard (one might almost say hackneyed) puzzle, so to make things more interesting we could have the brass key start out a bit bent, so that the player has to work out some way to straighten it before it'll work. In that case we want to give the player a fairly strong hint that it's nonetheless the right key. We can achieve all this by giving the **brassKey** a custom **isBent** property and checking for it in trunk's **lockOrUnlockAction** method (the action method of a **LockWith** command simply calls **lockOrUnlockAction(true)** while that of an **UnlockWith** command simply calls **lockOrUnlockAction(nil)**).

TADS 3 Tour Guide

```
trunk : KeyedContainer, Heavy 'large black trunk' 'large black trunk' @mainCave
  initSpecialDesc = "A large black trunk rests in the middle of the cave. "
  initiallyLocked = true
  keyList = [brassKey]
  lockOrUnlockAction(lock)
  {
    if(gIobj.isBent)
      reportFailure('{The iobj/he} fits the lock but won\'t quite turn in it. ');
    else
      inherited(lock);
  }
;

brassKey : Key 'small brass key' 'brass key' @groundCave
  "It<<isBent ? ' looks slightly bent' : '\s been straightened'>>. "
  isBent = true
;
;
```

We now need to find a way to straighten the brass key so that it'll open the trunk. Among things players might try is hitting the key with various objects, or trying to put in the flame of the torch, and we should probably provide suitable responses to such attempts or even make some of them work. But for now, we'll adopt a more exotic solution, a futuristic Autorectifier (a device that straightens any bent device placed in it), which will eventually be discovered aboard a space station (once we've constructed the space station and a means of getting there). We've already defined the basic cylinder that's the core of the device, here's the complete definition along with its components:

```
autoRectifier : ComplexContainer 'silver cylinder' 'silver cylinder' @mainCave
  "It's about a foot high and five inches in diameter. A black ring surrounds
  the opening at one end. The only other feature of interest are a conspicuous
  orange button and the manufacturer's name inscribed just below the ring. "
  subContainer : ComplexComponent, SingleContainer { bulkCapacity = 3 }
  bulk = 4
  weight = 3
;

+ Component 'black ring' 'black ring'
  "The black ring appears to made of some kind of rubbery plastic, but the material
  is unfamiliar to you, as is its function. "
;

+ Component 'manufacturer\'s name' 'manufacturer\'s name'
  "According to the inscription this device was manufactured
  by ALDEBARAN AUTORECTIFIERS INTERPLANETARY CORP."
;

+ Button, Component 'orange button' 'orange button'
  "It's a large round button on one side of the cylinder. "
  dobjFor(Push)
  {
    action()
    {
      "When you push the button the cylinder starts to hum, and its interior glows
      with a light that starts orange, then changes to yellow, then finally
      a dazzling white, as the hum steadily rises in pitch. Suddenly the hum
      stops and the light inside goes out. ";
      foreach(local cur in autoRectifier.subContainer.contents)
        if(cur.isBent) cur.isBent = nil;
    }
  }
;
;
```

Note how we specify the contents of the cylinder in the button's actionDobjPush method, and that this method will set isBent to nil on anything placed in the cylinder (if it is not nil already); this will allow the device to be usable on any other bent objects we care to introduce into the game.

6.5. LockableWithKey

You may recall a little way back that we defined a [tardisKey](#) to be found inside the desk in the cabin. Where there's a key to a Tardis, there should be a Tardis somewhere, so we'll set about defining one and giving it a lockable door.

```
tardis : Enterable -> tardisDoor 'dark blue police box/tardis' 'Tardis' @hold
  "It's a small dark blue booth, with a blue light on the top and
  the words POLICE BOX above the door. "
  initSpecialDesc = "A dark blue police box stands in the corner. "
  specialDesc = "A Tardis, conspicuously disguised as a police box, stands here. "
;

+ tardisDoor : LockableWithKey, Door '(tardis) door' 'door'
  disambigName = 'Tardis door'
  keyList = [tardisKey]
;
```

For those not familiar with the BBC TV series "Dr Who", a Tardis is a type of time machine (the name is meant to be an acronym of Time And Relative Dimensions In Space). The main ability of a Tardis is to dematerialize at one point in time and space and rematerialize in another location in time and space. A Tardis is also larger inside than out. Finally, a Tardis is meant to blend in with its surroundings on rematerialization, but Dr Who's Tardis became stuck in its 1960s London appearance, and so always looks like a Police Box (a kind of dark blue phone booth) from the outside.

At this point we need to define the interior of the Tardis, so that there's somewhere to go to if the player attempts to enter it or go through its door. As is well known, a Tardis is bigger inside than out, so we could give it as many rooms as we liked. We'll stick to just two, but then there's the question of how to describe their relative positions. Compass directions won't mean much inside the Tardis, and it's not obvious that shipboard directions like port and starboard would mean much either. On the other hand, from the outside one would probably regard the side of the Tardis with the door as its front, so there would be some justification for regarding movement towards this door as "fore" and away from it as "aft", which means that shipboard directions might just about do. We don't want the customizations of the Cabin class aboard the Tardis though, so we'll simply use the ShipboardRoom class:

```
tardisControlRoom : ShipboardRoom 'Tardis control room' 'the Tardis control room'
  "The room is, of course, much larger than the Tardis looks from the outside.
  Its walls are white, with curious circular indentations.
  A white door leads out to the outside world (wherever or whatever that may be)
  and an inner door leads further into the vessel.
  At the centre of the control room stands a hexagonal control console. "
  fore = tardisDoorInside
  out asExit(fore)
  aft = tardisInnerDoor
  in asExit(aft)
;

+ tardisDoorInside : Lockable, Door ->tardisDoor 'outer white door*doors' 'white door';

+ tardisInnerDoor : Door 'inner door*doors' 'inner door';

tardisLivingQuarters : ShipboardRoom 'Tardis Living Quarters' 'the living quarters'
  "There's not much here at the moment, but a door leads out. "
  out = tardisLivingQuartersDoor
  fore asExit(out)
;

+ tardisLivingQuartersDoor : Door -> tardisInnerDoor 'door' 'door';
```

The beauty of the way `tardisDoorInside` points to `tardisDoor` is that whenever we move the Tardis to a new location, the player character will automatically emerge in that location on leaving the Tardis.

Once again, `LockableWithKey` is a mix-in class, so it should become before any Thing-derived class in an object or class definition.

6.6. Keyring

A KeyRing is a specialized BagOfHolding designed for use with keys. In practice this means that if a player character is carrying a KeyRing, every time he or she picks up a key it's automatically added to the KeyRing, and when he or she comes across a previous unencountered locked door, an attempt to unlock it will automatically cause every key on the KeyRing to be tried (until one fits).

Since this is a convenient object for the player to have, we'll let the player find one early on. The definition can be minimal:

```
Keyring 'silver (key) keyring/ring' 'silver keyring' @firstAidKit
;
```

To see how it works, you might like to try going through the game so far, picking up the silver keyring at an early stage, and seeing what happens when you pick up the two keys and try opening the locked door into the Tardis and the locked trunk without specifying which key to use.

There are not many methods or properties of Keyrings you'd generally want to override, but you might have cause to override **isMyKey(key)**. This method determines whether key is an acceptable object to be placed on the Keyring, and by default accepts anything of class Key. It may be, however, that you're developing a game with several types of key, for example the conventional metal rods with teeth that turn in cylindrical locks and flat magnetic cards that are pushed into slots, and you may feel that the same type of Keyring would hardly be suitable for both. If you defined a CardKey subclass for the second type of key you might want to define Keyrings for the two types of key thus:

```
conventionalKeyring: Keyring 'brass keyring' 'brass keyring'
    isMyKey(key)
    {
        return key.ofKind(Key) && !key.ofKind(CardKey);
    }
;

cardHolder: Keyring 'plastic cardholder' 'plastic cardholder'
    isMyKey(key) { return key.ofKind(CardKey); }
;
```

We'll be doing something like this presently, when we come to implement a [card key](#).

6.7. Openable

Most objects that are lockable are also likely to be openable - after all, there's not a lot of point in locking or unlocking an object that can't be open or closed. However, since most of the objects that are openable and closable tend to be container-like objects or door-like objects, in practice one tends to use classes like [Door](#) and [OpenableContainer](#) rather more than a bare Openable. We are not yet ready to introduce an example of a bare Openable in our test game, but there is one later, the [tardisPanel](#) object.

The Openable class inherits all the behaviour of [BasicOpenable](#) and the mainly adds handling (or at least additional preconditions) for a number of common verbs (OPEN, CLOSE, PUT IN, POUR INTO, LOCK, LOCK WITH, GET OUT OF, and BOARD).

Openable also defines **lockStatusReportable** to be (!isOpen); for a fuller explanation see [Lockable](#). Finally it defines an **openStatus()** method which returns a sentence like "it's open" or "it's closed" (without punctuation) which can be added to the description of an object to indicate whether it's open or closed. The library defaults are usually fine, but you may, for example, want to suppress "it's closed" either for aesthetic reasons or to disguise the fact that something is openable, in which case you might write something like:

```
openStatus { return isOpen ? inherited : ''; }
```

If you do that, however, the punctuation will look a bit wayward when the object is closed, so you also have to tweak the description of the object from something like:

```
"It's red and square. "
```

To

```
"It's red and square"
```

And then write your `openStatus` method thus:

```
openStatus = (isOpen ? ' . It\'s open' : '')
```

Note that the final full stop (or period) and space have been removed from the end of the object description and added instead to the start of the "It's open" message.

6.8. BasicOpenable

`BasicOpenable` is the base class for openable items. It defines the basic behaviour for objects that can be opened and closed, but no special handling for commands (such as `OPEN` and `CLOSE`) that might commonly be used for openable objects. It is much more likely that you will use subclasses of `BasicOpenable` (such as [Openable](#), [BasicDoor](#) and their subclasses) than `BasicOpenable` in game code. It is conceivable that you might want to subclass a custom kind of openable object from `BasicOpenable`, as it is conceivable that you might want to implement a `BasicOpenable` object in a game for an object that can be open and closed but not does respond to normal opening and closing commands (e.g. because it can only be opened and closed by pushing a button or pulling a lever), but these are left as exercises for the interested reader. The chief importance of `BasicOpenable` is that it defines the behaviour common to all its descendants. The important properties and methods to know about are:

- **initiallyOpen**: set this to true if you want the object to start out open. The default is nil.
- **isOpen()**: use this method to determine whether the object is open (true) or closed (nil), but do not overwrite this property in game code to make an object open or closed; call `makeOpen` instead.
- **makeOpen(stat)**: call this method to open or close the object programmatically, by calling `makeOpen(true)` or `makeOpen(nil)`. You can also override this method to bring about additional side-effects of opening or closing the object, but if you do so be sure to remember to call `inherited(stat)` somewhere in your overridden `makeOpen(stat)`.
- **openDesc()**: the method/property that provides an additional description to say whether the object is open or closed; the English library defaults are "open" and "closed", which are good enough for most purposes.

7. Light and Fire

7.1. Light and Fire - Introduction

So far we have to rely on the MEGA or FIAT LUX cheating (I mean debugging, of course) command in order to explore dark locations. The time has come to do the job properly by providing the player with light sources. These can be as simple as an object that permanently glows, or as relatively complex as candles, torches/flashlights or oil lamps, together with matches to light them.

7.2. brightness

The simplest way to provide portable light is to create a portable object and set its brightness property to a suitable level, e.g.:

```
Thing 'brass lantern' 'brass lantern' @mainCave
    "It's an ordinary brass lantern, except you can't turn it off. "
    brightness = 3
;
```

We shan't be making this lantern a permanent feature of the game, but if you want to experiment, by all means try it. You'll find that when the player character is carrying the lantern you can move around all the previously darkened areas easily, since the lantern now provides light. For some games in some situations this may be all you need. The other classes of light-providing objects we'll be looking at simply provide more sophisticated ways of adjusting the brightness property of (usually) portable objects.

7.3. LightSource

LightSource is the most basic class that provides some specialization for providing light. It provides a **makeLit(lit)** method for turning the light on and off, or, to be strictly accurate, for adjusting the brightness between the values of its **brightnessOn** and **brightnessOff** properties (by default 3 and 0 respectively). Again, by default, a LightSource starts out lit (i.e. with its **isLit** property set to true). It also describes itself as providing light when lit. It may sometimes be useful to use this class for lightsources other than the [Flashlight](#) and [Candle](#) types provided by the library. As an illustration, we can convert the brass lantern of the previous section into a curious device that is lit only when the player character is holding it.

```
LightSource 'brass lantern' 'brass lantern' @mainCave
    "It's an ordinary brass lantern, except it has no visible means of control. "
    isLit = nil
    moveInto(newCont)
    {
        makeLit(newCont == gPlayerChar);
        inherited(newCont);
    }
;
```

Once you've finished playing (experimenting seriously, I mean) with this lantern you can delete it; it's an impractical device and we shall have no further use for it in this game. In the meantime you may have noticed that an even simpler way to have defined it would have been:

```
LightSource 'brass lantern' 'brass lantern' @mainCave
    "It's an ordinary brass lantern, except it has no visible means of control. "
    isLit = (isHeldBy(gPlayerChar))
;
```

But I wanted to illustrate the use of the makeLit() method. The second way of defining it does, however, provide a useful illustration of the isHeldBy(actor) method, a method of Thing that returns true if the Thing is directly in the actor's inventory, but is not being worn by the actor (more or less).

7.4. Flashlight

A Flashlight is basically a LightSource with a switch that can be used to turn it on and off (just like a flashlight or, as we'd call it in Britain, a torch).

```
blackTorch : Flashlight 'large black flashlight/torch' 'large black torch' @mainCave
    "It looks a serious heavy-duty instrument, with a firm ridged grip and
    a powerful bulb. "
    brightnessOn = 4
    bulk = 2
    weight = 2
;
```

There's no particular reason for setting brightnessOn to 4 here, other than the fact that it's described as a powerful torch and to demonstrate that it can be done. You can try this torch/flashlight out if you like, but we won't be leaving it lying around in mainCave for the player to pick up so easily. Instead we'll put in a storage cabinet aboard the Tardis:

```
tardisLivingQuarters : ShipboardRoom 'Tardis Living Quarters' 'the living quarters'
    "These living quarters are pretty bare right now, but there is a storage cabinet
    fixed to one wall, and a door that leads out. "
    out = tardisLivingQuartersDoor
    fore asExit(out)
;

OpenableContainer, Fixture 'storage cabinet' 'storage cabinet' @tardisLivingQuarters
    "The large cabinet is painted a cream colour and looks securely fixed to the wall. "
;

+ blackTorch : Flashlight 'large black flashlight/torch' 'large black torch'
    "It looks a serious heavy-duty instrument, with a firm ridged grip and
    a powerful bulb. "
    brightnessOn = 4
    bulk = 2
    weight = 2
;
```

This, of course, leaves players with the problem of finding an alternative light source before they can reach the Tardis. We'll deal with that next.

7.5. Candle & FireSource

Candle is the other specialization of LightSource provided in the library (strictly speaking, it inherits from FueledLightSource, which in turn inherits from [LightSource](#)). It can, of course, be used to implement candles (as we'll do in just a minute), but it is useful for implementing any light source with a limited fuel supply or burn life (as we shall go on to explore).

We have already defined some candles in our game (back when we implemented a [Dispenser](#)). You may recall we in fact defined our own RedCandle class. As yet, however, we have provided no means of lighting them. What we need for the task is a FireSource. One is available in the shape of the flaming torch attached to the wall, but we need to make some changes to it so it will act as a FireSource.

```
FireSource, Fixture 'flaming torch torch/flame/flames' 'torch' @mainCave
    "The torch, which is fixed firmly to wall by a steel bracket, is blazing merrily,
    its flames casting a bright but flickering light over the cave. "
    cannotTakeMsg = 'It\'s fixed to the wall. '
    preCondIobjBurnWith = static inherited - objHeld
    isLit = true
    disambigName = 'flaming torch'
;
```

There's a few things to note here. Firstly, the default handling for a FireSource requires that the actor is holding it before it can be used to light anything else; this is not appropriate, or not even possible, for a flaming torch fixed to the wall, so we need to removed the objHeld precondition. This is what we do with preCondIobjBurnWith = static inherited - objHeld, which, so far as the compiler is concerned, is precisely the same as if we had written:

TADS 3 Tour Guide

```
iobjFor(BurnWith)
{
    preCond = static inherited - objHeld
}
```

Which might have been clearer, but is also a bit more long-winded. This shorter syntax is worth considering when overriding only a single method of a `dobjFor` or `iobjFor` set, but probably not worth the loss of clarity if overriding two or more such methods. (To construct the appropriate method or property name, begin with 'verify', 'check', 'action' or 'preCond', then add either 'Dobj' or 'Iobj' and conclude with the action specifier, e.g. 'Take', 'Drop', or, as here, 'BurnWith').

We need to set `isLit` = true, otherwise the `FireSource` will be regarded as unlit and thus not capable of lighting anything else (until lit itself). Finally, we add a `disambigName` in case there's ever a name clash with the torch (flashlight) we defined in the last section.

A further customization that might be worth making is to the vocabulary for the `BurnWith` command. That defined in the library allows you to BURN, LIGHT, IGNITE or SET FIRE something WITH something. In the case of the candle and the torch, however, it would be perfectly natural for the player to type, for example, LIGHT CANDLE FROM TORCH. To allow this, we can simply override the vocabulary defined in the library, leaving everything else as it is:

```
modify VerbRule(BurnWith)
    ('light' | 'burn' | 'ignite' | 'set' 'fire' 'to') singleDobj
    ('with' | 'from') singleIobj
    :
;
```

Although we have already defined a `RedCandle` class, we need to tweak it a bit. Firstly, once a candle is lit, there's no reason why it shouldn't be used to light other candles (or anything else that's ignitable), so we should add `FireSource` to its class list. Secondly, a candle is not a very efficient light source (we shall assume), so we'll reduce its `brightnessOn` to 2 (this may not make much practical difference here, but one could devise situations in which it could be made to). Thirdly, a `Candle` continues to burn until its **fuelLevel** reaches zero, its `fuelLevel` being decremented by 1 each turn. By default a `Candle` starts with a `fuelLevel` of 20, but we'll be less generous with our `Candles`, making them a temporary expedient until the player finds a more efficient light source. Finally, once a candle has burned down, there's nothing left but a stub, and this should be reflected in its description. It's this last requirement that will add most of the complication.

```
class RedCandle : FireSource, Dispensable, Candle 'red candle*candles' 'red candle'
desc()
{
    switch(fuelLevel)
    {
        case 0:
            "It's just the stub of a red candle. "; break;
        case 1:
            "There's not much of it left. "; break;
        case 2:
            "It's little more than a stub. "; break;
        case 3:
            "It's a short red candle. "; break;
        case 4:
            "It's reasonably long. "; break;
        default:
            "It's a long red candle. ";
    }
    if(isLit)
    {
        "It's alight";
        if(fuelLevel < 3)
            ", but its flame is starting to flicker";
        ". ";
    }
}
isEquivalent = true
isListedInContents = (!isIn(myDispenser))
myDispenser = candleBox
fuelLevel = 7
brightnessOn = 2
sayBurnedOut()
```


TADS 3 Tour Guide

```
{
    inherited;
    local cur = new RedCandleStub;
    cur.moveTo(location);
    moveTo(nil);
}

dobjFor(BurnWith)
{
    check()
    {
        if(fuelLevel < 1)
            sayBurnedOut;
        else
            inherited;
    }
}

;

class RedCandleStub: Thing 'red candle stub*stubs*candles' 'red candle stub'
    "It's just the stub of a red candle. "
    isEquivalent = true
    weight = 0
    bulk = 0
;
```

The desc() method provides a suitable description of the candle at various fuel levels, adding a suitable second sentence if it's lit. When the candle runs out of fuel its **sayBurnedOut()** method is called, which makes this a good place to add any other special handling for this eventuality. What we want here is for the original candle to be replaced by a worthless candlestub object, so we create a new RedCandleStub, move it into the location of the candle that's just burned down to nothing, then move the burned-out candle to nil to dispose of it. We can't just tinker with the name and vocabWords properties of the burned out candle object, since the **isEquivalent** property being set to true means that all members of the RedCandle class must be effectively identical. The checkDobjBurnWith() handles the case when the player douses the candle just before it goes out and then tries to light it again; the standard library behaviour would prevent the candle being lit, and so it would never be replaced with a stub.

Prior to version 3.0.9 this code a strange bug would occur if you pick up the box, take a candle and then try to light it. For some reason that is not entirely clear to me, this resulted in the objBurning precondition attempting to take and light a series of candles in turn from the box until a stack overflow occurred. TADS 3.0.9 corrects this bug, but the way to prevent it in earlier versions is to remove the objBurning precondition (which tries to light a fire source if it's not already lit) and instead simply to make it illogical to try to light something with an unlit fire source.

```
iobjFor(BurnWith)
{
    preCond = [objHeld]
    verify()
    {
        inherited;
        if(!isLit)
            illogicalNow('{The iobj/he} is not lit. ');
    }
}
```

But if you're using TADS 3.0.9 or above this should not be necessary.

7.6. OilLamp

Actually, there's no OilLamp class in the TADS 3 library, but that doesn't stop us creating an oil lamp of our own. We've deliberately made the candles rather annoying to use, so the player will need a steadier light source, even before s/he encounters the flashlight in the Tardis.

In fact, an oil lamp is virtually the same as a candle, in that when lit it gives off light, and continues to burn until extinguished by the player or until it runs out of fuel. We can therefore base our oil lamp on the Candle class. The main complication comes from the fact that, unlike a candle, an oil lamp can be refuelled, so we need to add handling

TADS 3 Tour Guide

for oil being poured into the lamp. We also need to provide a description that depends on the state of the lamp, and to have the lamp warn the player when it's about to go out. The last can conveniently be achieved in the `burnDaemon` method, which is called once every turn that the lamp is lit.

```
oilLamp : FireSource, Candle 'brass (oil) lamp' 'brass oil lamp' @sack
desc()
{
    "It's a fine, polished brass oil lamp, in good working order. ";
    if(isLit)
    {
        "It's currently lit";
        if(fuelLevel < 4)
            ", but the flame is starting to burn low";
        ". ";
    }
}
fuelLevel = 20
fuelCapacity = 100
burnDaemon()
{
    if(fuelLevel < 4 && fuelLevel > 0)
        "\nThe oil lamp is starting to burn low. ";
    inherited;
}
iobjFor(PourInto)
{
    verify()
    {
        if(fuelLevel == fuelCapacity)
            illogicalNow('The oil lamp is already full. ');
        if(isLit)
            illogicalNow('You can\'t pour fuel into the lamp while it\'s lit. ');
    }
    check()
    {
        if(gDobj.oilLevel == nil)
            failCheck('That\'s not appropriate fuel for an oil lamp. ');
    }
    action()
    {
        local fuelTransferred = min(gDobj.oilLevel, fuelCapacity - fuelLevel);
        fuelLevel += fuelTransferred;
        gDobj.oilLevel -= fuelTransferred;
        if(fuelLevel == fuelCapacity)
            "You refill the lamp. ";
        else
            "You pour some oil into the lamp. ";
    }
}
bulk = 2
weight = 2 + (4 * fuelLevel)/fuelCapacity
;
```

There's nothing very complicated in any of this. We include the check for `fuelLevel > 0` (as well as `fuelLevel < 4`) in the overridden `burnDaemon` to avoid being told that the lamp is burning low after it's gone out. The two checks in `verifylobjPourInto` prevent the player from filling a lamp that's already full, or pouring oil in while the lamp is lit, while `checklobjPourInto` won't allow anything to be poured into the lamp from an object that hasn't got a non-nil `oilLevel` property. The action method then calculates the amount of oil poured in, which is the lesser of the amount of oil in the source container and the amount needed to fill the oil lamp to capacity. This quantity is then added to the `fuelLevel` in the lamp and deducted from the `oilLevel` in the source container, following which an appropriate message is displayed to describe the action. Since one might expect the weight of the oil lamp to vary with the amount of fuel it contains, the weight property is calculated appropriately.

The obvious source for oil for the lamp is the oil can we have already defined. But to make it suitable as a source for the lamp, we almost have to redefine it totally. While we're at it, we'll also move it into the `smallCave`:

```
oilCan : Thing 'oil can/oilcan' 'can of oil' @smallCave
"The can is <<howFull()>>. "
howFull()
{
```

TADS 3 Tour Guide

```
if(oilLevel<1)
    "empty";
else if(oilLevel < maxOilLevel/10)
    "almost empty";
else if(oilLevel < maxOilLevel/3)
    "about quarter full";
else if(oilLevel < (2*maxOilLevel)/3)
    "about half full";
else
    "more or less full";
}
initSpecialDesc = "An old oil can lies abandoned on the ground. "
dobjFor(PourOnto) { verify() { } }
dobjFor(PourInto)
{
    verify()
    {
        if(oilLevel < 1) illogicalNow('The oil can is empty. ');
    }
}
dobjFor(LookIn) asDobjFor(Examine)
initializeThing()
{
    inherited;
    oilLevel = maxOilLevel;
}
oilLevel = 0
maxOilLevel = 500
weight = 1 + (10 * oilLevel)/maxOilLevel
;
```

This is perhaps less complicated than it looks. The lengthy howFull() method merely provides a description (or rough estimate) of the amount of oil left in the can. The verifyDobjPourInto makes it illogical to pour any more oil from the lamp when it is empty. The initializeThing method makes sure the can starts off full, and once again we calculate the weight based on the amount of oil left in the can. It would probably be an unnecessary complication to add any further handling for PourOnto, since the amount of oil involved in lubricating the latch on the sailors' locker (or anything else that we may decide needs lubricating) is likely to be negligible.

7.7. Matchstick & Matchbook

Although the oil lamp makes a much more convenient light source than the stock of candles, it could still be inconvenient if it went out some distance from a naked flame. We could make things easier for the player by providing a book of matches. Both the Matchbook (a subclass of Dispenser) and Matchstick (a subclass of LightSource and FireSource) are defined in the library, and can be used with very little customization. All we need to do is to provide a description for the Matchbook and a name and some vocabulary for our Matchsticks. If we wanted various different types of Matchstick in our game we might have to define subclasses of Matchstick to do it on, but in this case we can simply override the Matchstick class.

```
matchbook : Matchbook 'book matches' 'book of matches'
    @(dressingTable.subUnderside)
    "The matchbook bears a picture of a banana and the words
    CABAL LIGHTING CO. "
;

modify Matchstick
    vocabWords = 'match matchstick*matches'
    name = 'match'
    isEquivalent = true;
;

+ Matchstick;
+ Matchstick;
+ Matchstick;
+ Matchstick;
+ Matchstick;
+ Matchstick;
```

TADS 3 Tour Guide

```
+ Matchstick;  
+ Matchstick;  
+ Matchstick;  
+ Matchstick;
```

7.8. Dynamite

For some time now we've had that pesky boulder blocking the cave to the west of main cave, and we've had to allow the player to pick it up in the absence of any way to blow it up. Well, now the time has come to try for a more explosive solution.

Not only might a stick of dynamite look a little like a candle, in some ways it behaves a little like one, except that something a bit different happens when it burns down. We could probably thus make a reasonable stab at a stick of dynamite by adapting a Candle and overriding `sayBurnedOut()` for the explosion. The main complication is handling all the different situations that could arise when the dynamite explodes. To keep things manageable, we'll handle just four. If the player character is still holding the dynamite when it explodes, s/he's killed. If the player character is close enough to the dynamite to touch it when it explodes, s/he's still killed, but with a different message. If the dynamite explodes when it's in a position to destroy the boulder, the boulder is destroyed and the dynamite removed. That leaves the situation where the dynamite explodes when the player character is at a safe distance, but is not in the right place to destroy the boulder. If we simply removed the stick of dynamite this would leave the player character alive, but the game unwinnable, so we need instead to allow the player to retrieve it again (under the guise of another stick of dynamite). We shall simply ignore the question of what damage the explosion might do to any other objects in the game, since to implement it would probably be too complicated.

One thing at a time; first we need to amend the definition of the boulder:

```
+ boulder : Container, Heavy 'boulder/crack' 'boulder'  
    "The huge boulder is blocking the exit to the west; there seems to be a  
    small crack in it. "  
    bulkCapacity = 2  
;
```

We make the boulder a Container as well as Heavy so that we can insert the dynamite in it, and we describe it as having a crack to give the player a clear hint that that's what s/he's meant to do.

We'll give the dynamite a short fuse and not much illumination when lit, and an appropriate description. The complication comes in the `sayBurnedOut()` method. We'll first test for the player character holding the dynamite and kill him or her if s/he is. We'll then do the same but with a different message if the player character is in a position to touch the dynamite, which we can test with the dynamite's **canBeTouchedBy** method. We'll then see if the dynamite is in the boulder. If it is we'll replace the boulder with boulder fragments. Penultimately we'll move the Dynamite back into nil and reset its `fuelLevel` to 3 in case we need to reuse it.

The really tricky bit is how to let the player know that the dynamite has exploded. Since the `sayBurnedOut()` method is called by a SenseDaemon, it or any method called by it won't display any text unless the dynamite is in scope for the player character; but if the dynamite were in scope, the player character would already be dead. But we don't want to keep the player guessing how long it takes for the dynamite to explode. Since the fuse is so short we can virtually guarantee that the player character will still be in earshot when the dynamite explodes, so we don't want to get involved in complex sense path calculations or the setting up of lots of SenseConnectors and SensoryEvents. The simplest way to get round the sensory context set up by the SenseDaemon used on the Candle class is to use the **callWithSenseContext** function to set up a temporary, different sense context:

```
dynamite : Candle 'stick dynamite/fuse' 'stick of dynamite'  
    "It's a white cylinder with a short fuse. <<isLit ?  
    'The fuse is lit and burning down fast. ' : nil >>"  
    fuelLevel = 3  
    brightnessOn = 1  
    sayBurnedOut()  
    {  
        if(isHeldBy(gPlayerChar))  
        {  
            "The dynamite explodes with a mighty bang and blows your hand off. But  
            since you're killed by the blast you probably won't be needing it  
            any more.\b";  
            endGame(ftDeath);  
        }  
    }
```

TADS 3 Tour Guide

```
if (canBeTouchedBy(gPlayerChar))
{
    "The dynamite denonates close by, but you are killed by the blast almost
    before you hear the bang. ";
    endGame(ftDeath);
}
if (isIn(boulder))
{
    boulderFragments.moveInto(boulder.location);
    boulder.moveInto(nil);
}
callWithSenseContext (nil, nil, {"You hear a muffled explosion nearby. "});
moveInto(nil);
fuelLevel = 3;
}
;
```

If the first two parameters (source and sense) of `callWithSenseContext` are nil then we effectively creating a universal, unrestricted sense context, allowing whatever happens in the function forming the third argument to be sensed from anywhere. We use the short form of this function definition, which means precisely the same as if we had written `new Function { "You hear a muffled explosion nearby. "; }`. We could have avoided all this complication had we used a Fuse on the dynamite, instead of trying to adapt the Candle. We'll remodel the dynamite using a [Fuse](#) in due course.

Finally, we need to define the `boulderFragments` object that is to replace the boulder when the dynamite detonates:

```
boulderFragments : Decoration 'boulder larger fragments/chunks' 'boulder fragments'
    "Most of the fragments are tiny, though there are a few larger chunks.
    They are scattered everywhere. "
    inRoomDesc = "Fragments of a boulder are littered all over the cave. "
    isPlural = true
;
```

Note that this once again makes use of our custom `inRoomDesc` property (which adds itself to the room description). A game that hadn't implemented this device would probably have to use `specialDesc` instead.

There is still one task left to achieve, namely to provide the player with a way of finding the dynamite in the first place. We'll cover that as the first task in the next chapter. In the meanwhile, if you want to try it out, you can temporarily set the dynamite's starting location to somewhere convenient like `mainCave` - but don't forget to set it back again (to nil) before starting the next section.

Finally, for the `endGame()` function, see above on [cannotGoThatWayInDark](#), where we first defined it.

8. Hiding & Finding

8.1. Hiding & Finding - Introduction

Nothing is hidden except in order to be revealed, nor does anything become concealed but that it might come into the open. - Mark 4.22

Mark was writing about Jesus' parables, not Interactive Fiction, although since according to at least some commentators Mark treats Jesus' parables as riddles and according to Nick Montfort (*Twisty Little Passages*. Cambridge, MA & London: MIT Press, 2003) the riddle is of the precursors of IF, there may be a tenuous link here. It may be that Mark portrays Jesus' parables as employing concealment as a strategy of revelation; it is certainly the case that IF authors often hide objects in their games with the intention that the player will find them (hopefully with more success than the disciples in Mark).

There are various ways objects can be hidden and made to appear in response to a player action. We'll first explore how this can be done using Classes and concepts we've already met, and show how the library classes [Hidden](#) and [PresentLater](#) can help with the task.

8.2. Hiding with Words

Perhaps the most basic way you could go about trying to hide something is by obscuring the way it's described, at least until it comes into the player's possession. We want to make the diamond lying at the end of the path a bit less obvious. To do this we could juggle the various description and name properties of the diamond, giving it a null string `initSpecialDesc` so it isn't listed anywhere until moved, and employ our custom `inRoomDesc` method to add a little hint to the room description. As a first (and somewhat over-the-top attempt) we might try something like:

```
diamond : Thing 'sparkle' 'sparkle' @pathEnd
    "It looks like a genuine diamond - and a valuable one too, exquisitely cut
    and multifaceted. "
    iobjFor(CutWith) { verify() { } }
    initSpecialDesc = ""
    initDesc = "It's hard to make out; maybe there's something there, or
    maybe it's just a trick of the light. "
    inRoomDesc()
    {
        if(!moved) "Something seems to sparkle among the rocks. ";
    }
    moveInto(newDest)
    {
        if(!moved)
        {
            initializeVocabWith('sparkling diamond');
            name = 'diamond';
            "Take a sparkle? You'll be wanting to drink a rainbow next!\b
            Oh well, if you insist. So, you scrabble among the rocks with your
            clumsy little finger and - my goodness! They close upon the sparkle,
            and as you pick it up it turns out to be something very solid and
            hard after all - a diamond no less! ";
        }
        inherited(newDest);
    }
;

Decoration 'rocks' 'rocks' @pathEnd
    desc = (diamond.inRoomDesc)
    isPlural = true
;
```

This is really pretty horrendous, quite apart from the sarcasm of the message displayed when the player first takes the diamond, he or she is meant to guess that the correct action here is TAKE SPARKLE. This makes it little better than a "read the author's mind" puzzle. It can be improved, however, by adding handling for searching the rocks (which is something the player is more likely to think of):

TADS 3 Tour Guide

```
diamond : Thing 'sparkle' 'sparkle' @pathEnd
  "It looks like a genuine diamond - and a valuable one too, exquisitely cut
  and multifaceted. "
  iobjFor(CutWith) { verify() { } }
  initSpecialDesc = ""
  initDesc = "It's hard to make out; maybe there's something there, or
  maybe it's just a trick of the light. "
  inRoomDesc()
  {
    if(!moved) "Something seems to sparkle among the rocks. ";
  }
  moveInto(newDest)
  {
    if(!renamed)
    {
      "When you pick it up it turns out to be a gem - a diamond no less! ";
      rename();
    }
    inherited(newDest);
  }
  rename()
  {
    initializeVocabWith('sparkling diamond');
    name = 'diamond';
    renamed = true;
  }
  renamed = nil;
;

Fixture 'rocks' 'rocks' @pathEnd
  desc = (diamond.inRoomDesc)
  isPlural = true
  dobjFor(LookIn)
  {
    action()
    {
      if(diamond.moved)
        "There's not much here. ";
      else if(diamond.renamed)
        "A diamond nestles among the rocks. ";
      else
      {
        diamond.rename();
        "Hunting diligently among the rocks you come across the source of the
        sparkle - something tangible and hard - a diamond! ";
      }
    }
  }
;

```

This may be acceptable, but it's quite a lot of code for finding a diamond, and unless this is exactly what you want, there are probably better ways of doing it, as we shall see.

8.3. Finding by moving

Finding concealed items can be implemented in code using the classes and methods we have already seen, e.g. by moving an object from nil into the player character's location when it is discovered. This is the technique we'll use to find the dynamite. We'll assume that the dynamite is buried in the sandy floor of the small cave. Since it's not obvious that one should dig there, and the player can hardly be expected to try digging in every location in the game, we'll drop a heavy hint by leaving a spade leaning against the wall of the cave:

```
spade : Thing 'spade' 'spade' @smallCave
  "It's a good solid spade, with a stout wooden handle and sharp steel blade. "
  initSpecialDesc = "A spade leans against the wall of the cave. "
  verifyIobjDigWith { logicalRank(150, 'digging implement'); }
;

```

TADS 3 Tour Guide

Note that the last line of this definition is simply a more compact way of writing:

```
iobjFor(DigWith)
{
    verify() { logicalRank(150, 'digging implement'); }
}
```

Having defined the digging implement, we need to define a floor that can be dug in and will yield dynamite when dug. You will recall that we start the [dynamite](#) in limbo (i.e. location = nil). Once the dynamite is discovered, we do not want the player to find another stick, unless the first one has been destroyed (by exploding). When it is destroyed, the stick of dynamite is moved back into nil, which means the player could then dig up another one from the small cave floor. This is fine, since it prevents the game becoming unwinnable should the player allow his first stick of dynamite to detonate in the wrong place. It would be good, however, to indicate that the second (and any subsequent) stick of dynamite is 'another' one, and not the same one miraculously reconstructed (even though, in the internal implementation, that's precisely what it is). We'll want our sandy diggable floor to be a replacement for the defaultFloor in the cave, so we'll derive it from that:

```
smallCaveFloor : defaultFloor
    desc = "The floor of this cave is very sandy. "
    dobjFor(DigWith)
    {
        verify() { }
        action()
        {
            local another = (dynamite.moved ? 'another' : 'a' );
            if(dynamite.isIn(nil))
            {
                "You uncover <<another>> stick of dynamite. ";
                dynamite.moveInto(smallCave);
            }
            else
                "You dig in the sand for a while but find nothing. ";
        }
    }
;
```

Finally, we need to amend our definition of smallCave so that it incorporates our special floor:

```
smallCave : DarkRoom 'Small Cave' 'the small cave'
    "The only way out of this small, sandy cave is to the south. "
    south : TravelMessage
    {
        -> secretPassage
        "You walk south for quite some way down a long tunnel. ";
    }
    roomParts = static inherited - defaultFloor + smallCaveFloor
;
```

8.4. sightPresence & isListed

Perhaps an even simpler way to hide an object and then reveal it is to set its **sightPresence** and **isListed** properties to nil. The former effectively put the object out of scope for any commands (such as TAKE or EXAMINE) and the latter prevents it being listed in the room description until sightPresence is set to true. We'll see how this works by hiding another of our coded tablets in the squareCave:

```
stoneTablet : Tablet '(loose) stone tablet*tablets' 'stone tablet' @squareCave
    inscription = "O P E R A\nY N U E R\nS T E T S\nI N F E F\nP A N I C"
    weight = 16
    sightPresence = nil
    isListed = (moved)
;
```

You may recall that the way out of this cave is through a stone archway we defined some way back as example of an [ExitPortal](#). This would seem a good place to hide the stone tablet; the player will only learn of the tablet's existence if he or she examines this arch. We don't want the stone tablet listed separately until it's free of the arch, which is why we set isListed = (moved) here, but we'll want the player to be able to interact with the tablet once he or she has examined the arch, so we set sightPresence to nil initially and let the archway set it to true when it's examined:

TADS 3 Tour Guide

```
+ ExitPortal -> mainCave 'ashlar arch/archway' 'archway'
  "The archway is beautifully constructed from dressed stones. <<looseStone>> "
  looseStone()
  {
    if(!stoneTablet.moved)
    {
      stoneTablet.sightPresence = true;
      "One of them seems a bit loose. ";
    }
    else
      "There's a gap in the stonework where one of them is missing. ";
  }
;
```

There's one further complication we have to bear in mind with this method, however, and that is if we had put the stone tablet in a container (e.g. making the archway a container from which the tablet could be removed and then reinserted) we should also need to add `isListedInContents = (sightPresence)`, otherwise the tablet would announce its presence when its container was examined, even if this wasn't what we wanted.

In the case of the stone tablet, manipulating `sightPresence` and `isListed` works reasonably well, not least because we want them to become true under slightly different conditions. Again, with items that descend from `NonPortable`, which would not normally be listed anyway, manipulating `sightPresence` can often be the most efficient means of bringing something to light (as in the example of the [hole at the end of the fluid link](#), which we'll come to eventually). Otherwise, though, it is usually simpler to use the [Hidden](#) class, which we shall look at next.

8.5. Hidden

The `Hidden` class provides a very convenient means of temporarily hiding something you want discovered later by some action by the player. It's a subclass of `Thing` that adds a **discovered** property and a **discover** method. By default, a `Hidden` starts with its **discovered** property set to nil. The overridden **canBeSensed** method then hides the object from the player (making it effectively invisible) until **discovered** is set to true, which is what calling the **discover** method does.

Suppose we want to hide the brass key under the rug in the `roundCave`. One way we can do this is simply to add `Hidden` to `brassKey`'s class list and then override `rug.actionDobjForLookUnder` [= the action method in `dobjFor(LookUnder)`] to call `brassKey.discover` if the brass key has not yet been discovered:

```
brassKey : Hidden, Key 'small brass key' 'brass key' @roundCave
  "It<<isBent ? ' looks slightly bent' : '\s been straightened'>>. "
  isBent = true
;

rug : Immovable 'large rectangular chinese rug/pattern/leaves/dragons' 'Chinese rug'
  @roundCave
  "The rectangular rug is patterned in pastel colours, mainly turquoise round the
  edge and principally golds and browns within. The patterns consists mainly
  of leaves and dragons. "
  initSpecialDesc = "A Chinese rug covers the centre of the floor. "
  specialDesc = "The Chinese rug has been pulled over to one side of the cave. "
  cannotTakeMsg = 'You probably could roll the carpet up and drag it around,
  but you don\'t want to be encumbered with it. '
  dobjFor(Pull)
  {
    action()
    {
      if(moved)
        "You can't pull the rug any further, it's already at the edge of the cave. ";
      else
      {
        "Pulling the rug over to the edge of the cave reveals a square hole in the floor. ";
        moved = true;
      }
    }
  }
}
```

TADS 3 Tour Guide

```
actionDobjLookUnder()
{
    if(brassKey.discovered)
        "{You/he} find{s} nothing else under the rug. ";
    else
    {
        "Under the rug {you/he} find{s} a small brass key. ";
        brassKey.discover();
        addToScore(1, 'finding the brass key ');
    }
}
;
```

Although it's not strictly necessary here, I've taken the opportunity to slip in an example of the **addToScore()** function. As defined here, it will add one point to the player's score together with the explanation (should the player issue a FULL SCORE command) that the award is for finding the brass key. You should be aware that this function will increase the score every time it's called, so you want to make sure it can only be called once if you only want the score to be increased once (or use the `addToScoreOnce` method instead). Since in this case we can be sure that the `else` clause will only be executed once, it's safe to use `addToScore` here. We'll come to a full discussion of [scoring](#) later.

Another (and in this case, simpler and easier) way of hiding something under something else is to put it in an [Underside](#), normally the `subUnderside` of a [ComplexContainer](#). We could do this with the book of matches we defined earlier, then they won't be revealed until the player specifically orders LOOK UNDER DRESSING TABLE, at which point they'll automatically be revealed. To do this, all we need to do is to add `Hidden` to the matchbook's class list:

```
matchbook : Hidden, Matchbook 'book matches' 'book of matches'
    @(dressingTable.subUnderside)
    "The matchbook bears a picture of a banana and the words
    CABAL LIGHTING CO. "
;
```

8.6. PresentLater

The `PresentLater` class is slightly more complicated than the `Hidden` class, although it can be used in ways that look much the same to the player. The official purpose of a `PresentLater` is to define an object that is not yet where you place it. That is, you can give a `PresentLater` object a starting location where you want it to appear subsequently, but at the preinitialization stage this location will be stored in its **eventualLocation** property and the object moved into `nil`. At the point in your game when you want it to appear in its predefined location you call its **makePresent()** method. Thus we could have implemented hiding the brass key by making it a `PresentLater`, Thing (note that `PresentLater` is a mix-in class - it does not descend from `Thing` itself) instead of a `Hidden`, by calling `brassKey.makePresent()` instead of `brassKey.discover`, and by testing `if(brassKey.moved)` instead of `if(brassKey.discovered)`. The effect (in terms of the transcript seen by the player) would have been identical. The main difference is that by using `Hidden` the brass key was in `roundCave` all along (but not visible until discovered), whereas by using `PresentLater` the brass key would not have been in `roundCave` until `makePresent()` was called.

There is one important distinction to bear in mind here, though, and that is that calling `makePresent()` on a `PresentLater` sets its **moved** property to true (since `makePresent` calls `moveInto`) while calling `discover` on a `Hidden` does not (because a `Hidden` is not moved when it is discovered).

In a simple case like that of the brass key where `Hidden` will do the job, it's probably best to use `Hidden`, unless there's some reason why you actually want the object concerned to be moved rather than simply made visible. In more complex cases, however, `PresentLater` may be the better choice, since it has a number of other methods we have yet to explore, and which make it a rather more powerful class than might appear from what we have seen so far.

In addition to its `makePresent`, method, `PresentLater` has a **makePresentIf(cond)** method; this moves an object from `nil` to its `eventualLocation` if `cond` is true, otherwise it moves it into `nil`. We can illustrate this by adding a console to the Tardis control room; the console has a panel behind which is a shallow compartment. Effectively, the compartment is only present in the control room if the panel is open, so we could use `makePresentIf(cond)` to bring it into the control room when the panel is opened and back into `nil` (in this case, out of sight) when the panel is closed:

```
tardisConsole: Fixture '(tardis) hexagonal control console' 'console' @tardisControlRoom
    "The six-sided console stands in the middle of the room. Amongst its controls
    are a slider, a dial, and a big red button. Beneath these is
    a panel set in the side of the lower part of the console. "
;
```

TADS 3 Tour Guide

```
+ tardisPanel : Openable, Component 'panel' 'panel'
  makeOpen(stat)
  {
    inherited(stat);
    tardisPanelCompartment.makePresentIf(stat);
    if(stat)
      "Opening the panel reveals a shallow compartment behind. ";
  }
;

+ tardisPanelCompartment : PresentLater, RestrictedContainer, Fixture 'shallow compartment'
  'shallow compartment' "It's about four inches deep. "
  validContents = [fluidLink]
;

++ fluidLink : Thing 'fluid link' 'fluid link'
  "It's a long transparent tube, half full of mercury. "
;
```

We'll do more with the fluid link later. For the moment let's continue to explore PresentLater. Let's suppose that in addition to the compartment we want a notice on the back of the panel that's only visible when the panel is open. We can make this a PresentLater too, and we could simply add `tardisPanelNotice.makePresent(stat)` to the definition of `tardisPanel.makeOpen`. But there is an alternative if we want to make several objects present at once: we can use **makePresentByKey(key)**. To use this you need to define a `plKey` on each of the PresentLater objects you want controlled by the single statement; e.g. you could define `plKey = 'tardis'` on both the compartment and the notice. To make all objects whose `plKey` is 'tardis' present at once you would then call `PresentLater.makePresentByKey('tardis')`; note that this method is called on the PresentLater *class*. In this case, however, we need to go one better still and use the **makePresentByKeyIf(key, cond)** method:

```
+ tardisPanel : Openable, Component 'panel' 'panel'
  makeOpen(stat)
  {
    inherited(stat);
    PresentLater.makePresentByKeyIf('tardis', stat);
    if(stat)
      "Opening the panel reveals a shallow compartment behind and a notice on
      the back of the panel. ";
  }
;

++ tardisPanelNotice : PresentLater, Component 'notice' 'notice'
  "The notice reads:\bFor best results ensure the fluid link is full of mercury. "
  plKey = 'tardis'
;

+ tardisPanelCompartment : PresentLater, RestrictedContainer, Fixture 'shallow compartment'
  'shallow compartment' "It's about four inches deep. "
  validContents = [fluidLink]
  plKey = 'tardis'
;

++ fluidLink : Thing 'fluid link' 'fluid link'
  "It's a long transparent tube, half full of mercury. "
;
```

One final point about PresentLater, which we'll describe without illustrating in our game, is that it's possible to use this class for an object that *starts* present and subsequently disappears. To do this we set its **initiallyPresent** property to true. It can then be made to disappear by calling `makePresentIf(nil)`, and to reappear again by calling `makePresent`.

9. Gadgets & Controls

9.1. Gadgets - Introduction

By gadgets we mean devices that incorporate buttons, switches, levers and the like, but also other odd gadgetry of the game author's devising. In the present chapter we shall explore some of the library classes that can be used in gadgets, and then go on to construct a couple of gadgets (or gadget-like puzzles) out of other elements.

What (for want of a better general term) we might call the library's 'gadget' classes include:

[Button](#)

[Lever](#)

[SpringLever](#)

[OnOffControl](#)

[Switch](#)

[Settable](#)

[Dial](#)

[LabeledDial](#)

[NumberedDial](#)

9.2. Button

The Button is the simplest kind of gadget class. It is simply something you can PUSH or PRESS. By default it then simply displays a message saying "Click. ", but you'll nearly always want to override this to do something else. You do so by overriding the action method of its Push handling, i.e. either

```
myButton : Button 'button' 'button'
  dobjFor (Push)
  {
    action()
    {
      /* My button-push handling here */
    }
  }
;
```

Or the functionally identical but cosmetically more compact (though arguably less clear) equivalent:

```
myButton : Button 'button' 'button'
  actionDobjPush
  {
    /* My button-push handling here */
  }
;
```

This is so simple that we have already used a couple of buttons: the first to open the [HiddenDoor](#) in the main cabin, and the second in the implementation of the [Autorectifying](#) device. We'll continue to use buttons on our various contraptions, notably on the [Tardis](#) control console, but there's no need to give another specific example here.

9.3. LabeledDial

The time has come to set sail in our ship. The panel on its quarterdeck is described as having a hexagonal hole, a wheel, and a lever. In order to sail the ship the player has to insert the hexagonal crystal in the hole, set the wheel to a destination, and pull the autopilot handle. The ship has four possible destinations, namely the north, east, south and west shores of the lake. It starts by the north shore, so the player will obviously have to turn the wheel to one of the other destinations to set sail for the first time.

We have already implemented the hexagonal hole (as a [RestrictedContainer](#)), so the next task is to construct the wheel. This in essence will be a thinly disguised dial that can be turned to N, S, W or E, so we shall use a `LabeledDial` to implement it. We list its possible settings in the `validSettings` property (which should always contain a list of strings). By default `LabeledDial.isValidSetting(val)` determines whether `val` is a valid setting by checking that it appears in the `validSettings` list, after converting both to upper case for the purpose of the comparison (thereby making the check non-case-sensitive). This does not itself convert the `curSetting` (current setting) property to upper case, which is what we want here, so we need to override `makeSetting(val)` so that this conversion takes place. This conversion is desirable both because it looks better to have the game report "This shows that the wheel is currently turned to E" than "This shows that the wheel is currently turned to e" (even if the player types TURN WHEEL TO e), and because it slightly simplifies the other use we are going to make of the `curSetting` property here, namely to determine the direction and destination of travel that correspond to the current setting of the wheel:

```
++ wheel : LabeledDial, Component 'wheel/pointer' 'wheel'
  "It looks much like a traditional wooden spoked ship's wheel, but incorporates a
  pointer that indicates the four points of the compass. This shows that the wheel
  is currently turned to <<curSetting>>. "
  validSettings = ['N', 'S', 'E', 'W']
  curSetting = 'N'
  directions = ['north', 'south', 'east', 'west']
  destinations = [lakeRoom, southShore, eastShore, westShore]
  makeSetting(val)
  {
    curSetting = val.toUpper();
  }
  curDirection = (directions [ validSettings.indexOf(curSetting) ] )
  curDestination = (destinations [ validSettings.indexOf(curSetting) ] )
;
```

9.4. SpringLever

A `SpringLever` is a lever that returns to its original position after being pulled. It is functionally similar to a `Button`, except that it responds to PULL instead of PRESS or PUSH. Apart from giving `SpringLever` the usual vocabulary, name and description properties, all one need normally do is to override its `actionDobjPull` method with whatever one wants to happen when the lever is pulled. In this case we want nothing to happen unless the hexagonal crystal is in the hexagonal hole and the wheel points to a destination other than where the ship already is, so we first test for these cases. If the conditions for setting sail are met, we display a suitable message making use of `wheel.curDirection` (defined in the [LabeledDial](#) example above) to describe the direction of travel, and then move the ship into its new destination as defined by `wheel.curDestination`:

```
++ SpringLever, Component 'long silver (black) lever/knob/plaque' 'lever'
  "It's a long silver lever with a black knob on the end. A silver plaque
  screwed just underneath it is inscribed AUTOPILOT. "
  dobjFor(Pull)
  {
    action()
    {
      if(!hexCrystal.isIn(hexHole) || wheel.curDestination == ship.location)
        "Nothing happens. ";
      else
      {
        "A hum starts up deep in the bowels of the ship. With a reluctant creak,
        the capstan turns, drawing up the anchor. The ship judders, then starts
        gliding out into the middle of the lake. From there
        it continues <<wheel.curDirection>> until it comes to rest by the
        <<wheel.curDirection>>ern shore, where the capstan lowers the anchor,
        and the ship moors itself up, port side to the shore. ";
      }
    }
  }
;
```

TADS 3 Tour Guide

```
        ship.moveInto(wheel.curDestination);
    }
}
;
;
```

Before trying this out, it would be well to define the three new locations that can now be reached by the ship. If you are splitting your code into separate source files, you might like to start a fresh source file for each of these rooms, since each will be the start of a new region of the map:

```
eastShore : Room 'Stone Jetty' 'the stone jetty'
    "This bleak stone jetty is little more than a narrow corridor between the lake to
    the west and the rough cave wall to the east. A broad flight of stone steps leads
    down to the south, while a much narrower flight leads up to the north. "
;

westShore : Room 'Sandy Beach' 'the sandy beach'
    "For an underground lake this section of shore forms a surprisingly large beach. The
    lake laps the shore to the east, while a pair of paths lead up from the beach to the
    cave complex beyond, one to the northwest and the other to the southwest. "
;

southShore : Room 'Rocky shore' 'the rocky shore'
    "The rocky shore looks so barren here as to be scarcely worth visiting, apart
    from a narrow tunnel leading off to the south. "
;
;
```

Note: depending on when you last recompiled the game you may need to do a complete recompile (Build -> Full Recompile for Debugging) after adding these locations. Having done so, you can try sailing the ship round the lake.

9.5. Settable

Settable is the base class for the various types of Dial, like the [LabeledDial](#) we constructed a couple of sections back. It can also be used to construct any other kind of settable control we care to devise. Here we'll use it to implement a slider on the Tardis control console; after all, now we've got the ship to sail round the lake, it's time to see what we get the Tardis to do.

What we want to achieve is quite complicated. Ultimately the Tardis will be controlled by a combination of a slider that can be set to any letter from A to Z and a dial that can be turned to any number from 0 to 9. The Tardis's destination will be decided by a combination of both settings (making 260 in all), but unless the fluid link is refilled with mercury, the slider setting will have no effect. Some of the Tardis's destinations will be predefined (the interesting ones), while we'll dynamically create (rather boring) ones for when the player goes to an undefined destination. This makes it very hard for players to find the useful destinations by trying settings at random - instead they'll have to find the suitably planted list of useful destinations.

So much for the summary, now let's get on with the implementation. As promised, we'll first make the slider. Make sure the following code is positioned in your code so that the slider is a Component of the tardisConsole object:

```
+ tardisSlider : Settable, Component 'slider' 'slider'
    "The slider can be set to any letter of the alphabet.
    It's currently set to <<curSetting>>. "
    curSetting = 'A'
    makeSetting(val)
    {
        curSetting = val.toUpper;
    }
    isValidSetting(val)
    {
        val = val.toUpper;
        return (val >= 'A') && (val <= 'Z') && (val.length==1);
    }
    setToInvalidMsg = 'The slider can be set only to a single letter from A to Z. '
;
;
```

Once again we override **makeSetting** to convert a lower case entry to an upper case one. The only new factor here is the need to override the **isValidSetting** method to define what settings we'll accept. In this case we want to accept

TADS 3 Tour Guide

any single character setting between A and Z inclusive, so we test accordingly. Finally, we override the **setToInvalidMsg** to display a more meaningful message in the event of the player attempting to set the slider to an inappropriate setting, such as SET SLIDER TO 9 or SET SLIDER TO OMEGA.

As things stands, the only verb that can be used to set the slider is SET; players might legitimately try to MOVE, PUSH or PULL the slider to a new setting. To cater for this we'll expand the vocabulary for the SetTo action:

```
modify VerbRule(SetTo)
  ('move' | 'push' | 'pull' | 'set') singleDobj 'to' singleLiteral
  :
;
```

This might be a bit more liberal than we'd ideally like (e.g. since you can SET DIAL TO 7 you'll now also be able to PUSH DIAL TO 7, PULL DIAL TO 7 or MOVE DIAL TO 7 as well), but erring a little on the side of liberality in allowing player commands is probably no bad thing.

9.6. NumberedDial

A Numbered Dial is simply a dial that can be turned to a number of numerical (and only numerical) settings. The definition of the numbered dial on the Tardis control console is pretty straightforward. We need to use two new properties used on NumberedDial, **minSetting** and **maxSetting**, which, as their names suggest, contain the minimum and maximum numerical value to which the dial can be set. Again, the object must be placed in your code so that it's a Component of tardisConsole:

```
+ tardisDial : NumberedDial, Component '(tardis) control dial' 'dial'
  "The numbers round the dial run from 0 to 9; the dial is currently set to <<curSetting>>. "
  minSetting = 0
  maxSetting = 9
  curSetting = '0'
  disambigName = 'Tardis control dial'
;
```

Note the need for a disambigName and the extra vocabWords corresponding to it; if the scales are in the control room at some point (as they probably will be once the player has picked them up) there would otherwise be no way the player could refer to this dial in preference to that on the scales.

One thing in particular you need to watch out for on NumberedDials is that while minSetting and maxSetting are **numerical** properties, curSetting is a **string** property. This can easily catch you out if you expect curSetting to contain a number because it's a property of a NumberedDial. What curSetting in fact contains here is a string representation of the number. If for any reason you need its numerical value (e.g. for calculation or comparison purposes) you will need to convert it using the toInteger function, e.g. (if we imagine a situation with two numbered dials):

```
local overallSetting = 10 * toInteger(dial1.curSetting) + toInteger(dial2.curSetting);
```

We shan't be needing to do that in this game; instead we'll take a brief (or perhaps not-so-brief) diversion from the gadget classes to construct a custom gadget for moving the Tardis round the universe in time and space.

9.7. Dynamic Locations

We now come to the complicated part, writing the code that will move the Tardis into a variety of locations (including dynamically-created ones) depending on the setting of the slider and the dial we've just defined. We'll start by defining the button the player has to press to set the Tardis in motion. Two rules will be enforced before the Tardis is moved: first the outer door to the Tardis must be closed, and secondly the fluid link must be in place inside its compartment. If the fluid link is not full, the slider will be treated as being set to 'A' regardless of its actual setting. We then query another object (to be defined below) to determine the Tardis's destination based on the setting of its two controls, and move the Tardis into its new location. Again, remember to place this definition in your code so that it's a Component of tardisConsole:

TADS 3 Tour Guide

```
+ tardisButton : Button, Component 'big red button' 'big red button'
  dobjFor (Push)
  {
    check()
    {
      if(tardisDoorInside.isOpen)
      {
        "A red warning message flashes up on the console:\n
        <FONT COLOR=RED>DOOR OPEN - TRAVEL ABORTED</FONT><.p>";
        exit;
      }
    }
  }
  action()
  {
    if(!fluidLink.isIn(tardisPanelCompartment))
    {
      "Nothing happens. ";
      return;
    }
    "The central control column pumps up and down with the strange wheezing
    sound that only a superannuated TARDIS can make, and you feel the weird
    sensation of being translated along unfamiliar dimensions. After about
    half a minute, it all stops. ";
    local destcode = (fluidLink.full ? tardisSlider.curSetting : 'A' )
      + tardisDial.curSetting;
    local dest = tardisDestinations.destination(destcode);
    tardis.moveInto(dest);
    if(dest == outsideCave) entranceTunnel.blocked = true;
  }
}
```

The reason for the final line of code is to ensure that the tunnel is blocked when the Tardis arrives outside the cave, even if the player hasn't previously triggered the rockfall by climbing the ladder up from the main cave. Otherwise, the game could become unwinnable (since if the player character left the Tardis outside, entered the cave and climbed down the ladder, climbing the ladder again would trigger the rockfall, rendering the game unwinnable).

At this point, you might want to add a full property to the fluid link:

```
++ fluidLink : Thing 'fluid link' 'fluid link'
  "It's a long transparent tube, half full of mercury. "
  full = true
;
```

We'll change this definition in due course, but having the fluid link always full for now will give you access to all the possible Tardis destinations while you're testing the Tardis drive.

Next we need to define the `tardisDestinations` object that will work out where to send the Tardis based on the settings of the dial and the slider. There are potentially 260 locations keyed by a combination of letter and digit (e.g. A0, A2, C9, T5). We shan't actually want to fill all these potential slots, but we do want an easy way of knowing what destination corresponds to what slot (defined by a letter and digit combination). This is a good job for a **LookupTable**. A `LookupTable` contains a set of pairs of arbitrary values, one of which is the key, and the other the value corresponding to the key. To add such pair to the table (or modify the value corresponding to an existing key) you simply use a statement of the form `myTable[key] = value`; To find the value corresponding to a given key you use `value = myTable[key]`; We want the Tardis to start with a number of preset destinations, so we make `tardisDestinations` a `PreinitObject`, which means that its **execute** method will be called during preinitialization. We use that `execute` method to define our preset destinations with statements like `destinations['A0'] = hold`; (the starting location of the Tardis).

If all the Tardis was going to do was visit these preset destinations, then all we'd need to do is to return `destinations[destcode]` to any calling routine wanting to know the destination corresponding to `destcode`. But the player will probably try several `destcodes` for which there is no entry in the destinations `LookupTable` (e.g. 'K2'), and for which `destinations[destcode]` would therefore be `nil`. We have to decide how we will handle these cases. We could just disallow travel in these instances, but we'll attempt something much more interesting: we'll create a new location on the fly and add it to the table. However (just to complicate things still further!) we may want to limit the number of dynamically-created locations that can be spawned by this means. To achieve this we'll add a `destinationsCreated`

TADS 3 Tour Guide

property to keep track of how many we've created and a `maxDestinationsToCreate` property to hold the maximum number of destinations we'll let be created. Rather than imposing a sharp cut-off when `destinationsCreated` reached `maxDestinationsToCreate`, however, we'll gradually reduce the probability of creating a new locations till it falls to zero when `destinationsCreated` hits the maximum. To achieve this we compare `rand(101)` (which returns a random number between 0 and 100 inclusive) with `destinationsCreated` as a percentage of `maxDestinationsToCreate`. If `rand(101)` is less than or equal to this percentage, we abort the creation of a new location and simply return `nil`. Clearly, once this percentage reaches 100, we'll always abort, but when it's very low, we'll nearly always go ahead and create a new location.

Before we get there, however, we should check whether there's an existing location corresponding to `destcode`. If there is, we simply return it. If there isn't, and we pass the probability test, we create a new destination, add it to the table of destinations, and return it to the calling routine (in `tardisButton.actionObjPush`). To create the new location we simply do what we'd do to create any new object dynamically, call `new` plus the classname, in this case `new TardisDestination`. We'll leave `TardisDestination` the job of defining itself:

```
+ tardisDestinations : SecretFixture, PreinitObject
  destinations = static new LookupTable
  execute()
  {
    destinations['A0'] = hold;
    destinations['A2'] = spaceStation;
    destinations['C9'] = redDesert;
    destinations['T5'] = outsideCave;
  }
  maxDestinationsToCreate = 50
  destinationsCreated = 0
  destination(destcode)
  {
    local dest = destinations[destcode];
    if(dest != nil) return dest;
    if(rand(101) <= (destinationsCreated * 100)/maxDestinationsToCreate )
      return nil;
    else
    {
      dest = new TardisDestination;
      destinationsCreated ++;
    }
    tardisDestinations.destinations[destcode] = dest;
    return dest;
  }
;
```

There was strictly speaking no need to make `tardisDestinations` a `SecretFixture` as well as a `PreinitObject` here, but it is convenient to fit it into the containment hierarchy (notionally as another part of the `tardisConsole`). An alternative would have been to have made `tardisDestinations` purely a `SecretFixture`, and to have defined its destinations (or called a custom method which did so) within its `initializeThing` method.

We next need to go on to define the `TardisDestination` class. This will be a subclass of `OutdoorRoom`. All we'll create with it is a set of fairly barren and boring outdoor locations, but it would be good to provide some superficial variation between them. To do this we'll use `TardisDestination`'s **construct** method, which is called whenever a dynamic object is created with the `new` statement. In this particular `construct` method we'll randomly assign an epithet, a colour and a terrain which in combination will make up the name of the room, and use those together with a randomized sky description and a random synonym of 'stretches' to vary the room description. Finally, we'll want to give the player the illusion of being able to wander around, so we'll point the cardinal direction properties to a series of `FakeConnectors` (which we'll define separately rather than trying to create dynamically). The definition of the `TardisDestination` then becomes:

```
class TardisDestination : OutdoorRoom
  construct()
  {
    epithet = rand('vast','barren', 'desolate', 'empty', 'lonely' , 'spooky', 'dead', 'grim');
    colour = rand('ochre' , 'red', 'green' , 'black' , 'brown', 'blue' , 'grey', 'white');
    terrain = rand('plain', 'forest', 'wilderness', 'swamp', 'jungle', 'tundra', 'desert',
                  'grassland' , 'wasteland', 'prairie');
    name = '^'+ epithet + '^'+ colour + '^'+ terrain;
    stretches = rand('stretches', 'extends', 'rolls', 'ranges', 'spreads', 'reaches');
    sky = rand('clear blue sky', 'threatening dark clouds', 'weird orange sky',
              'lurid green heavens', 'fluffy white clouds', 'black thunderclouds', 'reddening sky');
    inherited();
  }
}
```

TADS 3 Tour Guide

```
desc = "This <<epithet>> <<colour>> <<terrain>> <<stretches>> as far as the eye can see
      in all directions, one direction looking much like another under the <<sky>>. "
destName = ('the ' + epithet + ' ' + colour + ' ' + terrain)
north = wanderNorth
east = wanderEast
south = wanderSouth
west = wanderWest
epithet = nil
colour = nil
terrain = nil
stretches = nil
sky = nil
;
```

One point to note here is the use of the **construct** method; this is a special method that's called on any object created dynamically through a new statement at the point of creation. Note that a constructor can take parameters like any other method, in which case you would include them in the new statement. For example, instead of having TardisDestination constructor randomly choose an epithet, colour and terrain for itself, we could have these passed as parameters to the constructor:

```
class TardisDestination : OutdoorRoom
  construct(myEpithet, myColour, myTerrain)
  {
    epithet = myEpithet;
    colour = myColour;
    terrain = myTerrain;
    ...
  }
  ...
;
```

Then you could have set up a particular kind of dynamic TardisDestination with a statement like:

```
new TardisDestination('broad', 'purple', 'steppe');
```

However, in this game, we shall stick with the way we've done it. Of course it would be possible to elaborate this to add further variety still, but the above definition suffices to show the principle. Next we need to define the four FakeConnectors that all members of the TardisDestination will use:

```
wanderNorth : FakeConnector { "You wander off to the north for a while, but finding
                              nothing of interest you eventually turn round and come back. " }

wanderEast : FakeConnector { "You stride confidently off to the east, but the further
                              you go, the more it all looks the same, so after a while you
                              retrace your steps. " }

wanderSouth : FakeConnector { "The further south you go, the more you wish you hadn't
                              bothered, so in the end you give it up as a bad job and head back. " }

wanderWest : FakeConnector { "You carry on westwards as far as your legs will carry you,
                              but eventually you are forced to rest. Having rested, you decide you
                              really don't want to go any further that way, so you return the way
                              you came. " }
```

Again, these really all say the same thing in slightly different words, but some illusion of variety may be created thereby.

Two of the preset destinations have already been created, the hold (in which the Tardis starts) and outsideCave (where we need to return to win the game - having completed various other tasks). We need to define the other two (we'll eventually be adding more than two, but two will do for now, just to demonstrate the principle). First, we'll add the space station just as a stub to be filled in later:

```
spaceStation : Room 'Space Station - Observation Deck' 'the observation deck'
;
```

Then we'll add the redDesert, which we'll teasingly make resemble the randomly generated locations. We'll reward the player who bothers to explore with another of the mysterious tablets, however, although we'll make it the least valuable one so that players who don't find it won't miss too much.

TADS 3 Tour Guide

```
/* The Red Desert World*/
```

```
redDesert : Room 'Vast Red Desert' 'the vast red desert'
    "This huge red desert stretches to the horizon in all directions, all directions
    looking much the same under the brassy sky, except that a faint trail leads east. "
    north = wanderNorth
    west = wanderWest
    south = wanderSouth
    east = redRavine
;

redRavine : Room 'Narrow Red Ravine' 'the narrow red ravine'
    "The faint trail from the west comes to the end in this narrow red ravine. Apart
    from the narrow path leading east, rocky hillsides tower up on every side. On
    the south side is a narrow hole, which perhaps leads into a cave. "
    west = redDesert
    south = redCave
    in asExit(south)
;

+ Enterable ->redCave 'narrow (south) hole/cave/rockface' 'narrow hole'
    "The narrow hole in the south rockface looks just big enough to squeeze through. "
;

redCave : DarkRoom 'Red Cave' 'the red cave'
    "There is just enough room to stand in this small cave; the dark, rough red
    walls press in on every side, and the roof rapidly dips to meet the floor
    at the rear. The only way out is through a narrow exit to the north. "
    north = redRavine
    out asExit(north)
;

plasticTablet : Tablet 'plastic tablet*tablets' 'plastic tablet' @redCave
    inscription = "S M I L E\nP I P E R\nR O X L N\nA N T L E\nT R U S S"
    weight = 1
    initSpecialDesc = "A plastic tablet lies on the floor towards the rear of the cave. "
;
```

There's still one more task we need to perform before recompiling the game and trying all this out. The way we have defined `tardisDestinations.destination` means that it could return nil, especially if a lot of dynamic destinations have been created already. This in turn means that pressing the big red button could end up sending the Tardis into nil (via the `tardis.moveTo(dest)` statement in `tardisButton.actionDoObjFor(Press)`). This in itself is not too problematic - unless the player tries to leave the Tardis while it's in nil, which he or she could very well try to do. This *does* need to be prevented, since once the player character is moved into nil, there is nothing the player character can do to get out it, other than type UNDO, which is a bit mimesis-breaking. What we need to do is to prevent the player leaving the Tardis if it's in nil, providing a convincing (or at least reasonably plausible) reason for doing so. This is quite easy to do, since leaving the Tardis requires travel via a Door, and a Door is a type of TravelConnector, and we can simply override its `canTravelerPass` and `explainTravelBarrier` methods to disallow travel when the Tardis is in nil and explain why we're doing so:

```
+ tardisDoorInside : Lockable, Door ->tardisDoor 'outer white door*doors' 'white door'
    canTravelerPass(traveler) { return tardis.location != nil; }
    explainTravelBarrier(traveler)
    { "You can't go out: the Tardis hasn't materialized properly and there's nothing
      out there but the grey limbo, where nothing can exist. "; }
;
```

Now at last you should be in a position to recompile the game and take your Tardis for a spin.

9.8. Lever

A Lever is a Thing that has two states, pulled and pushed, represented by its **isPulled** property being either true or nil respectively. The Lever class adds specialized handling for the PUSH, PULL and MOVED commands. A PULL command is considered illogical for a Lever whose `isPulled` is true, and a PUSH command illogical if it is nil (once a lever has been pulled it's in its pulled position and can't be pulled again until it's been pushed back to its starting position). If a PULL or PUSH command passes the verification stage, the action method calls **makePulled(true)** or **makePulled(nil)** accordingly. By default this simply sets the `isPulled` property to the value of the parameter passed to

TADS 3 Tour Guide

the `makePulled` method, but it can be overridden to do far more interesting things, as we shall shortly see. A `MOVE` command is effectively translated into a `PUSH` or `PULL` command, depending on the current state of the `Lever` (`PUSH` if `isPulled` is true, `PULL` otherwise).

There may obviously be cases where you want something that is quite obviously and explicitly a lever, and the `Lever` class clearly simplifies the definition of such objects. In general, all you need do is override `makePulled(pulled)` to define the particular effects of pulling or pushing the lever and then call `inherited(pulled)` for the default handling. Since a lever is so obviously something that should be pushed or pulled, however, here we'll make at least some attempt to disguise it. We'll make it an apparently decorative feature on a stone altar in the temple (in case you're wondering "What temple?", we'll be defining it shortly). When the lever (thinly disguised as a banana-shaped projection from the north side of the altar) is pulled, a secret panel behind the altar slides open; when it is pushed the panel slides shut again. The complication is that the banana-shaped lever won't budge at all unless the weight of objects placed on the altar comes to exactly fifty-four pounds (information the player can discover by deciphering all those tablets we keep scattering about the place). This is how we'll do it:

```
stoneAltar : Fixture, Surface 'stone altar' 'stone altar' @temple
  "The altar comprises a massive stone slab, carefully carved and dressed into a
  smooth surface, apart from a curious banana-shaped projection at one end. "
  weight = 0
;

+ Lever, Component 'banana-shaped banana shaped projection' 'banana-shaped projection'
  "Protruding from the north side of the altar, the banana-shaped projection is
  its only decorative feature. "
  makePulled(pulled)
  {
    if(stoneAltar.getWeight != 54)
    {
      reportFailure('It won\'t budge. ');
      exit;
    }
    else if(pulled)
      "With a loud grating sound, the wall behind the altar grinds open. ";
    else
      "When you push the lever, the wall behind the altar grinds shut. ";
    inherited(pulled);
    templeWestWall.makeOpen(pulled);
  }
  weight = 0
;
```

Perhaps the only real subtlety here is making the weight of both the altar and the projection zero. The reason for doing this is that both weights (by default each 1) are included in the calculation of `stoneAltar.getWeight`, which we are using to check the weight of items placed on the altar. We could have compensated for this by adding 2 to the weight we were checking for, but doing it the way we've done it is almost certainly less confusing and less error-prone.

With a bit more tweaking in the `makePulled` method, the `Lever` class can be used for something even less obviously less like a lever. In a nearby location we'll put a gold statue standing on a gold plinth. If the statue is pushed it topples over revealing a cavity in the plinth, which cavity contains yet another tablet: appropriately, the golden one. Here's how we can implement the statue as a `Lever`:

```
goldenGrotto : DarkRoom 'Golden Grotto' 'the golden grotto'
  "The walls of this grotto glitter with gold dust embedded among the rock, but
  if there was any gold of any consequence here, it has long since been removed,
  apart from "
  southeast = westShore
  out asExit(southeast)
;

+ statue : Lever, Fixture 'gold statue' 'gold statue'
  "The gold statue depicts a solemn, regal figure of noble bearing wearing a
  golden crown. The figure's right hand looks as if it is clutching something
  that is no longer there. <<isPulled ? nil : 'The statue has been toppled off
  its base and is lying on the ground.'>> "
  inRoomDesc = "a golden statue <<isPulled ? 'standing proudly on a golden plinth'
  : 'lying on the ground'>>. "
  makePulled(pulled)
  {
    if(pulled)
    {
```

TADS 3 Tour Guide

```
"You aren't strong enough to pull the statue back upright. ";
    exit;
}
else
{
    "The statue topples over, revealing a cavity in the plinth beneath. ";
    plinth.initializeVocabWith('cavity');
}
    inherited(pulled);
}
isPulled = true
;

+ plinth : Container, Fixture 'plinth' 'plinth'
    "The plinth is a <<isOpen ? 'hollow' : 'solid'>> block of gold inscribed with the
    words <q>King Benedict the Banana-Bearer</q>. "
    isOpen = (!statue.isPulled)
;

++ goldTablet : Tablet 'gold tablet*tablets' 'gold tablet'
    inscription = "T F Q Z P\nN W O B E\nA U O U A\nF L O U R\nS T O P S"
    weight = 32
    feelDesc = "It feels mighty heavy! "
;

Decoration 'cavity' 'cavity';
```

In case you're wondering, the purpose of the seemingly pointless and actually locationless Decoration object at the end is to ensure that 'cavity' is a word the game recognizes even before the statue is pushed over, in case the player tries to refer to the cavity before the statue is TOPPLED; in such a case 'You see no cavity here' is a more appropriate response than 'The word "cavity" is not necessary in this story' (since the word "cavity" will be added to the dictionary words referring to the plinth once the statue is toppled).

We'll give [yet another example of a lever](#) when we're ready for it. To tidy up this part of the game, we need to create the temple and the connections between various locations. To start with we'll create the routes to the grotto and the temple from the shore of the lake:

```
westShore : Room 'Sandy Beach' 'the sandy beach'
    "For an underground lake this section of shore forms a surprisingly large beach. The
    lake laps the shore to the east, while a pair of paths lead up from the beach to the
    cave complex beyond, one to the northwest and the other to the southwest. "
    southwest = graveyard
    northwest = goldenGrotto
;

graveyard : DarkRoom 'Graveyard' 'the graveyard'
    "There is something decidedly eerie about this underground graveyard with its
    musty old tombstones. This is truly a place of death; nothing
    lives here, for this place never sees the sun; a dusty path leads off to the
    northeast and a strange, stone temple is situated just to the west. "
    northeast = westShore
    west = temple
;

+ Fixture 'musty old tomb/tombs/tombstones/tombstone' 'tombstones'
    "One in particular catches your eye, perhaps because of its curious inscription:
    \b<FONT FACE='TADS-Typewriter'>O 1 + + +\n8 R 2 + +\n+ 7 D 3 +\n
    + + 6 E 4\n+ + + 5 R</FONT>\b"
    isPlural = true
;

+ Enterable -> temple 'strange stone temple/door/lintel' 'temple'
    "It's a curious structure, seemingly carved out of the solid rock in an approximation
    to a gothic design. An inscription on the door lintel suggests that the temple is
    dedicated <q>to the unknown god</q>. "
;
```

There's nothing new in any of this. Of some interest, however, is the description of the tombstones, since this provides the key to deciphering the tablets (how this works should be rather more obvious when you see the description displayed). Next we need to define the interior of the temple:

TADS 3 Tour Guide

```
temple : DarkRoom 'Inside the Temple' 'the temple'
    "This gloomy temple looks like something out a gothic horror movie. The long, bare
    nave is populated only by a series of grim stone columns festooned with cobwebs.
    A large stone altar stands at the west end, "
    out = graveyard
    east asExit(out)
    roomParts = static inherited - defaultWestWall
    west = templeWestWall
;

+ CustomFixture 'grim (stone) column/columns' 'columns'
    "Four pairs of the stone column flank the central aisle, each column a
    grotesque, twisted shape, mocking the overall classical arrangement. "
    isPlural = true
    cannotTakeMsg = 'Moving these columns might be a seriously bad idea,
    since they appear to be holding up the roof; fortunately there\'s
    not the remotest prospect of your being able to shift any of them
    by so much as a nanocubit. '
;

+ Decoration 'cobwebs/webs/web/cobweb' 'cobwebs'
    "Multiple cobwebs festoon the space around the tops of the columns and
    the ceiling, but the spiders responsible have long since departed. "
    isPlural = true
;

+ templeWestWall : SecretDoor, defaultWestWall
    desc()
    {
        if(isOpen)
            "Most of the wall behind the altar has moved aside, leaving an aperture into
            a chamber beyond. ";
        else
            "The wall behind the altar is carved with strange patterns. ";
    }
    destination = templeChamber
    inRoomDesc = "behind which is a <<isOpen ? 'large open aperture in the
    wall' : 'stone wall carved with strange, abstract symbols'>>. "
;

++ Component 'strange abstract patterns/symbols/squares' 'symbols'
    "Some of them could almost be bananas, but most are spirals and squares. Several
    of the squares are subdivided into twenty-five smaller squares. "
    isPlural = true
;
```

The main thing to note here is how we handle the west wall of the temple. We remove the defaultWestWall from the temple's roomParts, but on this occasion we don't add our customized west wall back in. The main reason for this is because it makes use of our custom inRoomDesc property to add a description of itself to the room description, and for this to work it must be defined as in the temple's contents, not its roomParts. Finally, we need to define the secret chamber that's revealed when the west wall is opened:

```
templeChamber : DarkRoom 'Small Square Chamber' 'the small square chamber'
    "The most noticeable feature of this otherwise featureless chamber is the
    way out to the east. "
    out = temple
    east asExit(out)
;
```

That completes what is necessary for you to be able to compile and test the game once more. To try out the altar puzzle, you'll need to put the gold, stone and brass tablets there plus either an item weighing 2 or two items each weighing 1 (e.g. the torch and the brass key). You are, of course, entirely welcome to use the scales in the galley to try to find some other combination of objects coming to 54 pounds weight in total!

Of course, you may find it a bit tedious to go through the process of having to collect the right objects to put on the altar if you need to get into the small chamber beyond during the process of game development and testing (as you shortly will), so perhaps the time has come to define another magical debugging command. We'll call this one FORCE OPEN or FOPEN for short; if you FOPEN anything that's openable, it'll open up straight away, bypassing any locks or other inconvenient impediments:

TADS 3 Tour Guide

```
DefineTAction(ForceOpen)
;

VerbRule(ForceOpen)
  ('force' singleDobj 'open') | ('force' 'open' singleDobj)
  | ('fopen' singleDobj)
  :ForceOpenAction
  verbPhrase = 'force/forcing (what) open'
;

modify Thing
  verifyDobjForceOpen { illogical('{The dobj/he} {is}n\'t openable. '); }
;

modify BasicOpenable
  dobjFor(ForceOpen)
  {
    verify() { if(isOpen) illogicalNow('{The dobj/he} {is} already open. '); }
    action()
    {
      isLocked = nil;
      isOpen = true;
      "With a loud bang, {the dobj/he} flies open. ";
    }
  }
;
```

Don't forget to put code like this between `#ifdef __DEBUG` and `#endif`, so it doesn't end up in the released version of your game.

9.9. Dial

The Dial class is an immediate descendant of the [Settable](#) class and the immediate ancestor of the [NumberedDial](#) and [LabeledDial](#) classes. As such, it is not likely to be used much in game code, since if something is a dial at all, it is likely to be either numbered or labeled if it is to be at all useful, and if one wants something more general than a NumberedDial or a LabeledDial the chances are it'll be a Settable, like the slider we implemented before.

There may be one or two niches Dial can fill in game code, however. The most obvious might be a Dial with case-sensitive labels (since the LabeledDial assumes that its labels are not case sensitive). Although there may not be much call for case-sensitive labels, it could be possible to construct a minor puzzle of sorts out of a Dial with such labels, and it's probably easier (or at least, no harder) to do so starting from the Dial class than the LabeledDial class. Basically, all one need do to make it work is to define one's own `validSettings` list and override the `isValidSettings` method on such a dial thus:

```
isValidSetting(val) { return validSettings.indexOf(val) != nil; }
```

To show how this might work in practice, we'll add one of these in the secret chamber behind the stone altar, which we'll transmogrify into a lift in the process. The dial will be the lift control, which can be turned to thinly-disguised versions of the words UP and DOWN; no other setting has any effect. If the handle is turned to DOWN (disguised as Dupe OWN) the lift descends (unless it's already at the bottom). If it's turned to UP (disguised as dUPe own) the lift ascends again (unless it's already at the top):

```
templeChamber : DarkRoom 'Small Square Chamber' 'the small square chamber'
"The most noticeable feature of this otherwise featureless chamber is the
way out to the east, "
out = temple
east = (out)
moveInto(newDest)
{
  if(out == newDest)
    "and nothing happens. ";
  else
    "and with a shudder, the small chamber <<newDest==temple ? 'ascends' :
'descends'>> a long shaft, and finally comes to a halt. ";
  out = newDest;
}
;
```

TADS 3 Tour Guide

```
+ Dial, Fixture 'curious (wall) dial/handle' 'dial'
  "The dial comprises a handle set into the stone work that can be turned to
  point to any of the eight phrases carved round its circumference: <<listSettings>> .
  It currently points to <i><<curSetting>></i>. "
  inRoomDesc = "but there is also a curious dial set into one wall. "
  disambigName = 'curious wall dial'
  curSetting = 'DUPE OWN'
  listSettings()
  {
    stringLister.showList(validSettings);
  }
  validSettings = ['dupe own', 'dupE own', 'dUPe own', 'DUPe own',
                  'Dupe OWN', 'DUPE oWn', 'DUPE oWN', 'DUPE OWN']
  isValidSetting(val) { return validSettings.indexOf(val) != nil; }
  makeSetting(val)
  {
    if(val == curSetting)
    {
      "The dial already points to <i><<val>></i>. ";
      return;
    }
    "You turn the dial to <i><<val>></i>, ";
    switch(val)
    {
      case 'dUPe own': templeChamber.moveInto(temple); break;
      case 'Dupe OWN': templeChamber.moveInto(templeCellar); break;
      default: "but nothing happens. ";
    }
    inherited(val);
  }
}
```

Note that we have to change the definition of `templeChamber.east`, since the `asExit` macro won't do what we want if it refers to a directional property we subsequently change. Note also that we have overridden `templeChamber.moveInto` to something quite different from what `moveInto` normally does; we could have used any method name we liked for this, but `moveInto` is quite descriptive here, and we have no use for the standard `Thing.moveInto` handling here, so we might as well replace it. Finally, note that we define a `listSettings` method to avoid (the tedious and possibly error-prone process of) writing out the possible settings of the dial twice. This calls the convenient `stringLister.showList()` method to format the list in the form "a, b, c and d". Unfortunately neither the `stringLister` nor its `showList` method exists in the TADS 3 library, so we'll need to define them for ourselves:

```
stringLister : object
  showList(lst)
  {
    local lstLen = lst.length;
    for(local i=1; i<= lstLen; i++)
    {
      if(i not in (1, lstLen)) ", ";
      if(i == lstLen && lstLen > 1) " and";
      if(i>1) " ";
      say(lst[i]);
    }
  }
}
```

If you don't mind, or actually prefer, a comma after the penultimate item, you could probably simplify this a little.

Finally, we need to define the location the list/chamber arrives at if the dial is turned to the *Dupe OWN* position. As an attempt at brevity we might write:

```
templeCellar : DarkRoom 'Cellar beneath Temple' 'the cellar beneath the Temple'
  "This long, damp cellar probably hasn't been visited in years. "
  lift = templeChamber
  west = (lift.out == self ? lift : nil)
}
```

The `west` property shows a possible but not very sanitary method of checking that the lift is in place before allowing entry to it. It works after a fashion, but will cause an immediate runtime error if you then added, for example:

TADS 3 Tour Guide

```
in asExit(west)
```

Although we probably won't be implementing any other way into the temple cellar than via the lift, so the lift will always be available to the west when the player character is in the cellar, if we want to make this kind of test at all, it would be far better to use the recommended method we have already seen (using a nested `TravelConnector`), even though it is a bit more long-winded:

```
templeCellar : DarkRoom 'Cellar beneath Temple' 'the cellar beneath the Temple'
  "This long, damp cellar probably hasn't been visited in years. "
  lift = templeChamber
  west: OneWayRoomConnector
  {
    destination = (lexicalParent.lift)
    canTravelerPass(traveler)
    {
      return lexicalParent.lift.out == lexicalParent;
    }
    explainTravelBarrier(traveler)
    {
      "There's nothing that way but an empty shaft. ";
    }
  }
;
```

The above definition could be made a bit less verbose if we dispensed with the `lift` property and replaced each instance of `lexicalParent.lift` with `templeChamber`.

9.10. OnOffControl

An `OnOffControl` is an immediate descendent of `Thing` that responds to `TURN ON` and `TURN OFF` commands. It has an `isOn` property which keeps track of whether it is on (`isOn = true`) or off (`isOn = nil`), and a **`makeOn(val)`** method that simply sets `isOn` to `val`, but which can be overridden to incorporate the side-effects of turning the control on or off.

To complete our tour of gadgets and controls we'll return to the space station that we left in the most minimally-defined state. We'll use an `OnOffControl` to open the only door out of the room in which the Tardis materializes, at the same time filling in a few more details of the location:

```
spaceStation : Room 'Space Station - Observation Deck' 'the observation deck'
  "Judging by the huge observation window, this station is orbiting a huge
  blood-red planet. On the bulkhead is an electronic calendar, and underneath
  the calendar is a small green switch. "
  in = ssDoor1
;

+ OnOffControl, Fixture 'small green switch/maglock' 'small green switch'
  "The switch is labelled MAGLOCK. "
  makeOn(val)
  {
    ssDoor1.makeLocked(val);
    inherited(val);
  }
  isOn = true
;

+ ssDoor1 : IndirectLockable, Door 'steel door' 'steel door'
  inRoomDesc = "Next to both is a sliding steel door, which is currently <<isOpen ?
  'open' : 'closed'>>. "
  makeLocked(stat)
  {
    if(isLocked != isOpen)
    {
      "The door slides <<stat ? 'shut' : 'open'>>. ";
      makeOpen(!stat);
    }
    inherited(stat);
  }
;
```

TADS 3 Tour Guide

```
+ Fixture 'electronic calendar' 'electronic calendar'
  "According to the calendar the date is <<getDate>> "
  getDate()
  {
    local gt = getTime();
    local date;
    date = toString(gt[3]) + '-' + monthName(gt[2]) + '-' + toString(gt[1] + 1100);
    return date;
  }
;

function monthName(x)
{
  return ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'] [x];
}
```

The important code here is first of all in the `makeOn` method of the `OnOffControl`, which calls the `makeLocked` method of the door, and then in the `makeLocked` method of the door, which makes unlocking the door open it and locking it close it again. The electronic calendar is largely decorative, but it does serve to show the player that the Tardis has traveled well into the future. The `getDate()` routine makes the calendar display a date exactly 1100 years in the future from whatever the current (real world) date happens to be, and in a format that will be understood the same way on both sides of the Atlantic.

9.11. Switch

Switch is a simple extension of the generic [OnOffControl](#) that can be used with a SWITCH command without specifying ON or OFF, and treats FLIP synonymously. SWITCH X or FLIP X thus turns X on if it was off and vice versa.

We'll use a Switch in the second room of the space station as the apparent (but only apparent) duplicate of the `OnOffControl` we used in the first. When the player confidently tries to turn it off in the expectation of opening another door, however, s/he'll find it doesn't quite work as expected:

```
ssLivingQuarters : Room 'Space Station - Living Quarters'
  "These living quarters look totally abandoned; it doesn't look as if anyone has
  been here for years. A table is bolted to the centre of the floor.
  An open doorway leads through one bulkhead, while a closed sliding door is
  set in the opposite one. "
  out = ssDoorway
;

+ ssDoorway : ThroughPassage -> ssDoor1 'open doorway' 'open doorway'
  "The open doorway leads back to the observation deck. "
;

+ ssTable: Fixture, Surface 'table' 'table'
  "It's a plain steel table, bolted to the floor. "
;

+ ssDoor2 : IndirectLockable, Door 'door' 'door'
  "The door is a solid sheet of steel. It looks like it's meant to slide open.
  Next to the door is a small green switch. "
;

+ Switch, Fixture 'small green switch/maglock' 'small green switch'
  "The switch is labelled MAGLOCK. "
  makeOn(val)
  {
    if(!touched)
    {
      mercury.makePresent;
      touched = true;
      "The switch spits and fizzles, and then starts oozing a silvery liquid
      which gathers in a small pool on the floor. ";
    }
    inherited(val);
  }
}
```

TADS 3 Tour Guide

```
    isOn = true
    touched = nil
;

+ mercury : PresentLater, RestrictedContainer 'silver silvery liquid/mercury/pool'
    'silvery liquid'
    "It's a dense, silver-coloured liquid. "
    dobjFor(Take)
    {
        action()
        {
            "{You/he} can't pick it up, it simply runs between {your} fingers. ";
        }
    }
    validContents = [syringe]
    specialDesc = "A small pool of silvery liquid lies on the floor near the door. "
;
```

If you recall far enough back, when we first introduced the fluid link we hinted that it would need refilling with mercury - this is how the player will come by it. The validContents property of the mercury also hints how the mercury is to be collected. We'll see to all that next.

As a second example of a switch we'll add a switch to the Tardis control panel to open and close the door (the description of the control panel mentions a switch, but we've yet to implement it). The following object should be located shortly after the definition of tardisConsole so that it's located in the console:

```
+ tardisSwitch : Switch, Component '(tardis) bright green switch' 'green switch'
    "It's bright green. "
    makeOn(val)
    {
        "As {you/he} flip{s} the switch the outer door <<val ? 'opens' : 'closes'>>. ";
        tardisDoorInside.makeOpen(val);
    }
    dobjFor(Push) asDobjFor(Switch)
    isOn = (tardisDoorInside.isOpen)
;
```

There's a couple of things to note here. First, we define dobjFor(Push) asDobjFor(Switch) so that PRESS SWITCH or PUSH GREEN SWITCH works the same as FLIP SWITCH or SWITCH SWITCH, namely turning the switch on if its off and vice versa. Secondly, instead of having the switch's makeOn method change the value of its isOn property, we have it call the Tardis door's makeOpen method to open or close the door (and display a suitable message). We then define the isOn property to reflect the door's isOpen property so the two stay in sync automatically.

9.12. Lever (2)

In order to collect the mercury that spits from the second switch in the space station and insert it into the fluid link, the player will need to use the syringe found in the first-aid kit. To fill and empty the syringe requires pulling and pushing its plunger, which makes its plunger a good candidate to be yet another Lever. To do this we need to make substantial changes to the syringe object and then add the plunger as a component:

```
++ syringe: Thing 'syringe/needle' 'syringe'
    "The syringe is a long plastic tube with a needle at one end and a plunger at the
    other. It is <<fluid==nil ? 'empty' : 'full of '+ fluid.name>>. "
    fill(liquid)
    {
        fluid = liquid;
    }
    fluid = nil
;

+++ plunger : Lever, Component 'plunger/knob' 'plunger'
    "It's a small piece of white plastic with a round knob. "
    makePulled(pulled)
    {
        if(pulled && syringe.isIn(mercury))
        {
            syringe.fill(mercury);
        }
    }
;
```

TADS 3 Tour Guide

```
"The plunger pulls the silvery liquid into the syringe. ";
syringe.moveTo(gActor);
}
if(!pulled && syringe.fluid != nil)
{
    "A jet of <<syringe.fluid.name>> spurts from the needle";

    if(syringe.isIn(tinyHole) && syringe.fluid == mercury)
    {
        ", filling the fluid link";
        fluidLink.full = true;
    }
    ". ";
    syringe.fill(nil);
}
inherited(pulled);
}
;
```

Although the syringe is only meant to be filled with mercury, we allow for the possibility of handling other fluids by giving it a fluid property to describe what particular fluid, if any, it contains. Most of the complicated handling goes in the makePulled method of the plunger object, which we add as a component of the syringe. Here, we check to ensure that mercury is indeed what the syringe contains, if indeed it contains anything, but only really add handling for that case. In default of adding handling for a FillWith command (which we'll leave as an exercise for the reader) we make the player first PUT SYRINGE (or NEEDLE) IN MERCURY and then PULL PLUNGER to fill the syringe, but just to make things a bit easier for the player we make the handling of PULL PLUNGER move the syringe back into the player's grasp (since PUT NEEDLE IN MERCURY will have effectively dropped it from the player's grasp, and this may not be immediately obvious to the player, who could all too easily walk off after filling the syringe without realizing that it was being left behind).

It will be apparent from syringe.makePulled that in order to fill the fluid link from the syringe we need first to insert the syringe in some tiny hole. This will be a hole at one end of the fluid link, a hole only accessible, or visible, when the link is removed from its compartment. At the same time, we have to ensure that the player cannot replace the link in its compartment while the syringe is still sticking out of the hole, so we need to amend both the compartment and the link, as well as adding a tinyHole object:

```
+ tardisPanelCompartment : PresentLater, RestrictedContainer, Fixture 'shallow compartment'
    'shallow compartment' "It's about four inches deep. "
    validContents = [fluidLink]
    plKey = 'tardis'
    notifyInsert(obj, newContainer)
    {
        if(newContainer == self && syringe.isIn(tinyHole))
        {
            "You can't insert <<obj.theName>> into <<theName>> while the syringe is
            sticking out of it. ";
            exit;
        }
    }
;

++ fluidLink : Thing 'fluid link' 'fluid link'
    "It's a long transparent tube, <<full ? 'full of mercury' : 'with just a bit of
    mercury in it'>>. Both ends are capped with some kind of shiny
    metal<<isIn(tardisPanelCompartment) ? nil : ', and at one end is a tiny hole'>>. "
    full = nil
    iobjFor(PutIn) maybeRemapTo(tinyHole.sightPresence, PutIn, DirectObject, tinyHole)
;

+++ tinyHole : Component, RestrictedContainer 'tiny hole' 'tiny hole'
    "The tiny hole in the end cap of the fluid link is protected by some kind of membrane
    to prevent the contents escaping. "
    validContents = [syringe]
    sightPresence = (!fluidLink.isIn(tardisPanelCompartment))
;
```

The overridden tardisPanelComponent.notifyInsert method does the job of preventing an attempt to return the fluid link to its compartment with the syringe still sticking in it. We change the description of the fluid link so it describes whether it is full or not and mentions the hole at its end provided the hole is visible, and we change the full property to be nil to

TADS 3 Tour Guide

start with. It is possible that the player may try to PUT SYRINGE IN FLUID LINK instead of PUT SYRINGE IN HOLE, and we allow this by remapping the former command to the latter provided the hole is visible. We do this with the maybeRemapTo macro, which carries out the remapping only if its first argument evaluates to true. Finally, we make the tinyHole visible or not by overriding its sightPresence property to be true if and only if the link is out of its compartment. In a case like this (where we want something descended from NonPortable to appear and disappear), this is probably the easiest and most efficient way of achieving the effect.

If you now recompile and test the game, you should not only be able to refill the fluid link, but you should find that until you do so, setting the slider on the Tardis control console to different values has no effect on the Tardis's destination. Since filling the fluid link is a bit of a rigmarole to go through when testing, you might want to add the following cheating short cut (for testing purpose) in your debug code (between #ifdef __DEBUG and #endif):

```
DefineIAction(FillFluidLink)
    execAction
    {
        fluidLink.full = true;
        "Well, even if the fluid link wasn't full before, it sure is now! ";
    }
;

VerbRule(FillFluidLink)
    ('fill' 'fluid' 'link') | 'ffl'
    : FillFluidLinkAction
    verbPhrase = 'fill/filling the fluid link'
;
```

9.13. A Card Lock

We'll finish off the space station section by adding another lockable compartment, but this time one that uses a card key to open it. First we'll leave the card key lying around to be picked up - but we'll also damage it:

```
class CardKey : Key;

cardKey : CardKey 'white plastic card' 'white plastic card' @spaceStation
    "It's a piece of white plastic, about 80 x 30mm, with some blue letters
    printed on it that are now too indistinct to read. <<isBent ? 'Unfortunately,
    it also looks a bit bent. ' : nil>>"
    initSpecialDesc = "On the floor lies a white plastic card. "
    isBent = true
;
```

Now in the other accessible half of the space station we'll place the cabinet this key's designed to unlock, and put yet another of our tablets inside it. We'll make it a LockableWithKey, defining the cardKey as the key, but since the player may also try to insert the card, we'll remap a PutIn command to the appropriate UnlockWith command (so that PUT CARD IN SLOT is treated as UNLOCK CABINET WITH CARD):

```
ssCabinet : KeyedContainer, Fixture 'small cabinet/slot' 'small cabinet' @ssLivingQuarters
    "The front of the cabinet is flush with the bulkhead and contains a small slot. "
    inRoomDesc = "A small metal cabinet is set into another of the bulkheads. "
    keyList = [cardKey]
    keyIsPlausible(key) { return key.ofKind(CardKey); }
    initiallyLocked = true
    lockOrUnlockAction(lock)
    {
        if(gIobj.isBent)
        {
            reportFailure('{The iobj/he} won\'t fit in the slot. ');
            exit;
        }
        inherited(lock);
    }
    iobjFor(PutIn) remapTo(UnlockWith, self, DirectObject)
;
```

TADS 3 Tour Guide

```
+ silverTablet : Tablet 'silver tablet' 'silver tablet'
  inscription = "F R A N K\nE I I O I\nO T V N L\nF L E E T\nF O R H S"
  weight = 8
;
```

Note that we have introduced a new (to the reader) method of KeyedContainer (and other LockableWithKey objects), namely **keysPlausible**. This should return true if and only if a key might plausibly fit the lock; in this case the card key might but a conventional key obviously wouldn't. Only if keysPlausible(key) returns true for a certain key will that key be tried in an implicit action. The keyList property narrows down the list of keys that will actually operate the lock to the card key alone, but we override lockOrUnlockAction to prevent even this key from working if it is bent.

At this point we had best move our autoRectifier into its proper, futuristic, initial setting, so that the player has an immediate means of repairing the bent card key:

```
autoRectifier : ComplexContainer 'silver cylinder' 'silver cylinder' @ssTable
  "It's about a foot high and five inches in diameter. A black ring surrounds
  the opening at one end. The only other feature of interest are a conspicuous
  orange button and the manufacturer's name inscribed just below the ring. "
  subContainer : ComplexComponent, SingleContainer { bulkCapacity = 3 }
  bulk = 4
  weight = 3
;
```

If you move this definition in your source code, remember to move the component object definitions (the black ring, manufacturer's name, and orange button) with it. It probably won't take the player long to figure out that the autorectifier solves the problem of the bent card key, but this is then a reasonable clue that it might do the same for the bent brass key found elsewhere.

The introduction of the card key leaves us with one or two pieces of tidying up to do. First of all, a card key is not the sort of thing that should be added to our keyring, so we need to override its isMyKey method as envisaged when we first encountered the [Keyring](#) class:

```
Keyring 'silver (key) keyring/ring' 'silver keyring' @firstAidKit
desc() { }
descContentsLister = (examineLister)
isMyKey(key)
{
  return key.ofKind(Key) && !key.ofKind(CardKey);
}
;
```

Finally, we have a couple of lockable objects (the trunk and the Tardis door) for which the card key is quite obviously not the right kind of key. Rather than deal with them individually, it's easier simply to override keysPlausible on the class:

```
modify LockableWithKey
  keyIsPlausible(key) { return key.ofKind(Key) && !key.ofKind(CardKey) ; }
;
```

Since the ssCabinet object overrides this method in its own way, it won't be affected by this change.

10. Fuses & Daemons

10.1. Fuse

A Fuse is probably the simplest kind of Event to create. It simply fires an event after a certain number of turns. To set up a fuse you use a statement of the form

```
new Fuse(obj, &prop, turns);
```

Which means that after *turns* turns the method `obj.prop` will be executed (the `prop` method of the `obj` object), and then the fuse will be removed from the list of pending events. And that's basically it. You should note, of course, that in the argument list of the `new Fuse` call, the second argument is a property pointer, hence you need to use the `&prop` syntax. It's also helpful to know that if `turns = 0` the method will fire on the current turn, if it's 1 (one) it'll fire on the next turn and so on. And it may be worth while bearing in mind that the statement `new Fuse()` returns a pointer to the new fuse object created, so that if you want to refer to it subsequently, it's useful to use a statement like:

```
fuseID = new Fuse(obj, &prop, turns);
```

This makes it convenient to terminate the fuse prematurely, if for any reason you need to. To remove the fuse defined above from the list of pending events you would call:

```
fuseID.removeEvent;
```

This would not fire `obj.prop` prematurely, it would simply abort the fuse. If we had not used a `fuseID` property to store a pointer to the fuse, we could still abort it by calling:

```
eventManager.removeMatchingEvents(obj, &prop);
```

Where `obj` and `&prop` are the object and property pointer used in creating the fuse. This method returns true if any events (Daemons or Fuses) were found matching the specification and nil if not. It is thus not absolutely essential to store a pointer to the fuse if you may want to abort it, but it is probably more convenient. It can also be useful for keeping track of whether a fuse is active or not.

For our first example we'll return to the stick of [dynamite](#) we created earlier (something which quite literally has a fuse) and implement it with a fuse. It's still worth using the `Candle` class to implement the stick of dynamite, since the `Candle` implements a lot of handling (not least of the `BURN WITH` and `EXTINGUISH` commands) that's useful to us. Indeed, much of the original definition can stand, all we need to do is to override the `isLit` method to set up a Fuse (instead of the `SenseDaemon` the standard `Candle` employs); the changes are shown in bold:

```
dynamite : Candle 'stick dynamite/fuse' 'stick of dynamite'
  "It's a white cylinder with a short fuse. <<isLit ?
  'The fuse is lit and burning down fast. ' : nil >>"
  fuelLevel = 3
  brightnessOn = 1
  sayBurnedOut()
  {
    if(isHeldBy(gPlayerChar))
    {
      "The dynamite explodes with a mighty bang and blows your hand off. But
      since you're killed by the blast you probably won't be needing it
      any more.\b";
      endGame(ftDeath);
    }
    if(canBeTouchedBy(gPlayerChar))
    {
      "The dynamite denonates close by, but you are killed by the blast almost
      before you hear the bang. ";
      endGame(ftDeath);
    }
    if(isIn(boulder))
    {
      boulderFragments.moveInto(boulder.location);
      boulder.moveInto(nil);
    }
  }
```

TADS 3 Tour Guide

```
"You hear a muffled explosion nearby. ";
moveInto(nil);
fuseID = nil; // not strictly necessary here
fuelLevel = 3; // not strictly necessary any more

}
makeLit(lit)
{
    inherited(lit);
    if(lit)
        fuseID = new Fuse(self, &sayBurnedOut, 3);
    else
    {
        fuseID.removeEvent();
        fuseID = nil;
    }
}
;
```

The one slight oddity about this revised stick of dynamite is that however many times it extinguished and relit (before it actually detonates) its fuse always remains the same length - but perhaps in the context of the game that's just as well (the player always has three turns to get out of the way after lighting the fuse).

For our second example we'll create a Fuse and a Daemon on the same object, though for the moment we'll concentrate our attention on the Fuse. A solid gold tablet, such as we've placed in the plinth of the [statue](#), would be pretty heavy to carry around, and indeed it's the heaviest portable object (as defined by its weight property) that we've defined in the game. Rather than create an old-fashioned inventory puzzle by limiting the total amount of weight the player character can carry (so that, for example, the player character would have to drop everything else in order to carry the gold tablet, and arrange to distribute light sources along its path, which would all be pretty tedious) we'll limit the number of turns the player can carry the gold tablet before putting it down to take a rest. We'll use a [Daemon](#) to limit how long the player character can carry the gold tablet for, and a Fuse to simulate his or her recovery. We'll add a carried property to goldTablet which will hold the number of turns for which the tablet has been carried. Once this reaches 3 the player character becomes too tired and is forced to drop the tablet. At this point we set a fuse that is fired after three turns (provided the player leaves the gold tablet alone in the interim); when the fuse fires three turns later, it simply resets goldTablet.carried back to zero (which would allow the player character to pick up the gold tablet and carry it for another three turns).

In order to give the context, the following code shows the full redefinition of the goldTablet, but for now we'll concentrate only on explaining that for the fuse:

```
++ goldTablet : Tablet 'gold tablet' 'gold tablet'
    inscription = "T F Q Z P\nN W O B E\nA U O U A\nF L O U R\nS T O P S"
    weight = 32
    feelDesc = "It feels mighty heavy! "
    carried = 0
    afterAction()
    {
        if(daemonID == nil && isIn(gPlayerChar))
        {
            endFuse;
            daemonID = new Daemon(self, &daemon, 1);
        }
        else if(daemonID != nil && !isIn(gPlayerChar))
        {
            endDaemon;
            fuseID = new Fuse(self, &fuse, 3);
        }
    }
    daemonID = nil
    daemon
    {
        local carrier = self;
        while (!carrier.isDirectlyIn(gPlayerChar))
            carrier = carrier.location;
        gMessageParams(carrier);
        switch(carried++)
        {
            case 0: break;
            case 1: "{The carrier/he} {is} starting to feel very heavy. "; break;
            case 2: break;
        }
    }
}
```


TADS 3 Tour Guide

```
case 3: "You can't carry {the carrier/him} much further. "; break;
default: "You're forced to put {the carrier/him} down;
        it's too heavy for you. ";
        nestedAction(Drop, carrier);
    }
}
endDaemon
{
    if(daemonID != nil)
        daemonID.removeEvent;
    daemonID = nil;
}
fuse() { carried = 0; fuseID = nil; }
fuseID = nil
endFuse
{
    if(fuseID != nil)
        fuseID.removeEvent;
    fuseID = nil;
}
;
```

There's nothing magical about the names `fuse`, `fuseID` and `endFuse` we've given various methods and properties, we could have called them `tom`, `dick` and `harry`, it's just that the names we've given are a lot more meaningful. The `afterAction()` method tests whether anything has happened to change whether the gold tablet is still being carried or not. For reasons we'll explain more fully when we come to the daemon, the first if statement in this method tests for the tablet being picked up, and the second for its being dropped. If it's picked up we want to abort any fuse that's running (since the player has not rested from trying to carry the tablet that turn) so we abort the fuse with the `endFuse` method. This is a method we define ourselves; it does two things: first, if `fuseID` is not nil (i.e. if there is a fuse in operation) we terminate the fuse by calling `fuseID.removeEvent`, second we reset `fuseID` to nil to keep track of the fact that there's no longer a live fuse. The test for `fuseID` being nil makes it safe to call `endFuse` whether there's actually a fuse running or not; if `fuseID` is nil when `endFuse` is called `endFuse` has no effect.

Whenever the player ceases to carry the gold tablet (either by dropping the tablet itself, or by dropping something that directly or indirectly contains the tablet) we want to set a new fuse that will allow the player to recover after three turns (provide s/he doesn't try to pick the tablet up again during that time). We set the new fuse in the statement `fuseID = new Fuse(self, &fuse, 3)` which means that, unless we abort the fuse in the meanwhile, the method `self.fuse` (i.e. `goldTablet.fuse`) will be executed after three turns. Finally, all the `fuse()` method has to do is to reset `carried` to zero and reset `fuseID` to nil (so we have a convenient method of telling that the fuse is no longer active).

We'll now go on to explore how the [Daemon](#) works.

10.2. Daemon

A Daemon is only slightly more complicated than a [Fuse](#), in that while a Fuse fires a one-off event by executing a method after a set number of turns, a Daemon repeatedly calls a method at fixed intervals (unless or until the Daemon is terminated).

The syntax for setting up a Daemon is similar to that for creating a Fuse:

```
new Daemon(obj, &prop, interval);
```

This sets up a daemon that will call the method `obj.prop` every *interval* turns, where *interval* must be at least 1 (one). If *interval* is 1, `obj.prop` will execute each turn, starting with the current turn. If *interval* is 2, `obj.prop` will execute every other turn, starting with the next turn. In general, if *interval* is *n*, `obj.prop` will execute every *n* turns, starting in *n-1* turns time. Note once again that the second argument to the new Daemon call must be a property (or method) *pointer*, hence the `&prop` syntax.

As with a Fuse it's useful to store a reference to a Daemon when it's created, e.g.:

```
daemonID = new Daemon(obj, &prop, interval);
```

TADS 3 Tour Guide

This then allows the Daemon to be terminated with the command:

```
daemonID.removeEvent;
```

Otherwise to terminate the Daemon you'd need to call:

```
eventManager.removeMatchingEvents(obj, &prop);
```

Now to return to our example. You'll recall that the aim is to limit the number of turns for which the player character can carry the heavy gold tablet. The Fuse we've already looked at handles the player character's recovery; we'll use a Daemon to warn the player that the tablet is becoming an unbearable burden and then to enforce its dropping when the player has exhausted his or her allotment of turns:

```
++ goldTablet : Tablet 'gold tablet*tablets' 'gold tablet'
  inscription = "T F Q Z P\nN W O B E\nA U O U A\nF L O U R\nS T O P S"
  weight = 32
  feelDesc = "It feels mighty heavy! "
  carried = 0
  afterAction()
  {
    if(daemonID == nil && isIn(gPlayerChar))
    {
      endFuse;
      daemonID = new Daemon(self, &daemon, 1);
    }
    else if(daemonID != nil && !isIn(gPlayerChar))
    {
      endDaemon;
      fuseID = new Fuse(self, &fuse, 3);
    }
  }
  daemonID = nil
  daemon
  {
    local carrier = self;
    while (!carrier.isDirectlyIn(gPlayerChar))
      carrier = carrier.location;
    gMessageParams(carrier);
    switch(carried++)
    {
      case 0: break;
      case 1: "{The carrier/he} {is} starting to feel very heavy. "; break;
      case 2: break;
      case 3: "You can't carry {the carrier/him} much further. "; break;
      default: "You're forced to put {the carrier/him} down;
               it's too heavy for you. ";
               nestedAction(Drop, carrier);
    }
  }
endDaemon
{
  if(daemonID != nil)
    daemonID.removeEvent;
  daemonID = nil;
}
fuse() { carried = 0; fuseID = nil; }
fuseID = nil
endFuse
{
  if(fuseID != nil)
    fuseID.removeEvent;
  fuseID = nil;
}
;
```

The mechanics of controlling the Daemon are very like those of controlling the Fuse. We set up the Daemon and store a reference to it in the statement `daemonID = new Daemon(self, &daemon, 1)`, and we define a convenient method for terminating the Daemon in `endDaemon`, which calls `daemonID.removeEvent` provided `daemonID` is not nil, and then sets `daemonID` to nil so we can check that there's no longer a Daemon running (as with the `endFuse` method we defined earlier, this definition makes it safe to call `endDaemon` even if `daemonID` is nil and no Daemon is running). As in the case of the Fuse, there's nothing obligatory about the names `daemon`, `daemonID` and `endDaemon`, we could

TADS 3 Tour Guide

have called these properties and methods `boston`, `cambridge` and `worcester` (if we were particularly anxious name them after cities that are found in both England and Massachussets), but the names we have used make it a bit clearer what they're for.

The principal complication with what we're trying to do is that the restrictions on carrying the gold tablet should apply whether the player character is carrying directly, or in some other container such as a sack, or in a container within a container, such as a box in a sack, and so on. This means we cannot predict what sequence of actions will result the in tablet's being carried or no longer carried. It might be as simple as `TAKE GOLD TABLET`, or as long-winded as `PUT GOLD TABLET IN BOX; PUT BOX IN SACK; TAKE SACK`; or something even more convoluted. Or again, the player might `TAKE GOLD TABLET` and then `PUT TABLET IN SACK` when the player character (PC) is already carrying the sack. We need to try to find the simplest way handling all these eventualities.

One thing that helps us here is that the `whatsit.isIn(obj)` method returns true whether `whatsit` is directly in `obj` or is contained in something that's within `obj`, however deeply nested. Thus `goldTablet.isIn(gPlayerChar)` will be true whether the tablet is directly carried by the player character, or in a sack that the PC is holding, or in a box in a sack that the PC's carrying, and so on. Likewise `goldTablet.isIn(gPlayerChar)` will be nil whenever the PC is neither directly or indirectly carrying the tablet. We can thus use this in `goldTablet.afterAction` to check whether the PC is carrying the gold tablet or not after each turn in which the PC performs an action in the vicinity of the tablet. But we also need to know whether there's been a change of state, i.e. whether or not the PC was carrying the gold tablet before, since we don't want to keep spawning new daemons each turn that the gold tablet is carried and new fuses each time it is not. Fortunately we can check that by seeing whether there's a daemon already running (i.e. `daemonID` is not nil). If the PC is carrying the gold tablet and `daemonID = nil`, we need to create a new daemon (and kill any fuse that's running), because the PC must have started carrying the tablet this turn. Likewise, if `daemonID` is not nil and the PC is not carrying the tablet, s/he must have just ceased doing so, so we need to kill the daemon (and start a new fuse). *Because* we start a new daemon when the PC starts carrying the gold tablet and kill the daemon when the PC stops carrying the tablet, we can use the existence or otherwise of the daemon to check whether or not the PC was carrying the tablet the previous turn. The two checks we need in `afterAction` are thus `if(daemonID == nil && isIn(gPlayerChar))`, to determine that the PC has just started carrying the gold tablet, so that we need to start the daemon and stop the fuse, and `if(daemonID != nil && !isIn(gPlayerChar))`, to determine that the PC has just stopped carrying the tablet, so that we need to stop the daemon and start a new fuse running.

The `daemon` method (called each turn that the PC is carrying the gold tablet) uses a `switch` statement to display an appropriate message (or keep quiet), and finally to force the gold tablet to be dropped. The complication here is once again that the PC may either be carrying the gold tablet directly, in his hands, or in some other container, such as a box or sack. In the former case the player should be informed that it is the gold tablet that is becoming unbearably heavy, and eventually forced to drop the tablet; in the latter it would be better if the player were told that the box, sack or other carrier was becoming difficult to carry, and forced to drop the box or sack. The first four lines of the `daemon` method are thus devoted to identifying the object we want the rest of the routine to work with. We begin by declaring a local variable, `carrier`, and setting it to `self` (i.e. the tablet), which will be the object we want to refer to if the gold tablet is not inside anything. The while loop in the next two lines then walks up the containment tree until it finds an object that's directly held by the player (if the gold tablet is directly held by the player, it won't have much walking to do). After these three lines `carrier` will refer to whichever object it is that the player's directly holding, either the gold tablet itself, or whatever the gold tablet's being carried around in (if the tablet is inside a box which is inside a sack that's being carried, `carrier` will refer to the sack, not the box). The fourth line then uses the library macro `gMessageParams()` so that we can use `carrier` in parameter substitution strings when generating the messages in the `switch` statement.

To see how this all works out in practice, try recompiling and running the game. Then try carrying the gold tablet around first by hand and then in the sack. You should find that if you carry it by hand, you can just carry it into the temple and put it on the altar before having to put it down. Strictly speaking, this is all that *needs* to be done with it; the catch is that the player can hardly know this without first weighing the gold tablet, which will use up more turns than this even if the PC brings the scales to the tablet rather than the other way round. This doesn't stop the player completing the task, but it does force the PC to take a break from carrying the gold tablet at some point. If this was felt to be too much of a nuisance, one could perhaps go on to implement, say, a wheelbarrow object in which the tablet could be pushed around for longer distances (which would not count as carrying it). We'll do this when we come to look at the [TravelPushable](#) class.

10.3. SenseFuse

One problem you may have with a **Fuse** is that it could go off when the player character is not near enough to perceive the result, but that any text defined in the method that's executed when the Fuse fires will be displayed whether it describes something the player character could witness or not. This wasn't a problem with the Daemon and Fuse we defined on the gold tablet, but in other situations it might be. In such situations what you need is a **SenseFuse** - a special type of Fuse that won't display any messages if the player character isn't there to see (or hear, or smell) what happens.

The way to set up a SenseFuse is very similar to the way you set up a Fuse, except that there are a couple of extra properties:

```
new SenseFuse(obj, &prop, turns, source, sense);
```

This definition will cause `obj.prop` to be executed *after* `turns` turns, as with a Fuse. The difference is that the player will only see any messages displayed by `obj.prop` if, at that point in time, the player character can sense the *source* object (which in practice may often be the same as the `obj` object, but need not be) using the *sense* (which could be sight, sound, smell or touch - most likely one of the first two).

As an example we'll put a SenseFuse on an exploded bomb hidden under a pile of rubble. Once the player finds the bomb in the rubble, a new fuse is created that will cause the bomb to explode in three turns (killing the player if s/he is still rash enough to be around). Obviously, we'll also need to create an environment for the bomb, so we'll start by adding a new location the Tardis can reach:

```
+ tardisDestinations : SecretFixture, PreinitObject
  destinations = static new LookupTable
  execute()
  {
    destinations['A0'] = hold;
    destinations['A2'] = spaceStation;
    destinations['C9'] = redDesert;
    destinations['T5'] = outsideCave;
    destinations['Q7'] = londonStreet;
  }
  ...
;
```

Then we can proceed to define a couple of locations, some rubble, and a bomb:

```
/* London - 1940 */

londonStreet : OutdoorRoom 'City Street' 'the city street'
  "Several burned-out and half-destroyed buildings line this section of
  the city street along with the ones that are still standing. The destruction
  seems to have been fairly recent, since the rubble of fallen masonry still
  spills out onto the street, blocking the way south. The road continues to
  the north. "
  south : NoTravelMessage { "The rubble blocks your path. "; }
  east : NoTravelMessage { "The houses directly to the east are burned-out
  shells; they don't look safe to enter. "; }
  north = streetJunction
;

+ rubble : Immovable 'pile rubble' 'rubble'
  "The largest pile of rubble spills out into the street and blocks progress south.
  <<bomb.isIn(nil) ? specialDesc : nil>> "
  dobjFor(LookUnder)
  {
    action()
    {
      if(bomb.moved)
        "There's nothing much there but rubble. ";
      else if(bomb.discovered)
        "The bomb is still there. ";
      else
      {
        bomb.discover();
        "You find a metal cylinder buried among the rubble. It looks horribly
```

TADS 3 Tour Guide

```
        like a bomb. ";
    }
}
}
dobjFor(LookIn) asDobjFor(LookUnder)
specialDesc = "Pieces of rubble have been blown all over the street, surrounding
    a fresh bomb crater. "
specialDescOrder = 70
useSpecialDesc() { return bomb.isIn(nil); }
;

+ bomb : Hidden, Immovable 'unexploded bomb/cylinder' 'bomb'
    "It's a fat, round-nosed cylinder with tail fins, on a couple of which
        are painted tiny swastikas. "
    discover()
    {
        inherited;
        new SenseFuse(self, &explode, 3, self, sight);
    }
    explode()
    {
        "The bomb explodes, the blast sending chunks of masonry flying in all
            directions, one piece of strikes you square on the head. ";
        if(gPlayerChar.isIn(location))
            endGame(ftDeath);
        moveInto(nil);
    }
    cannotTakeMsg = 'You must be joking! '
    cannotPushMsg = 'That might set it off. '
    cannotMoveMsg = 'It\'s probably safest to leave it just where it is. '
;

streetJunction : OutdoorRoom 'Street Junction' 'the junction'
    "The street from the south meets another running east-west. A short way down
        to the street to the east a fire crew is fighting a blazing fire. "
    south = londonStreet
    east : FakeConnector { "After taking a few steps east you recall that
        discretion is the better part of valour and decide to keep out of the
        way of the fire crew. "}
    atmosphereList : ShuffledEventList

    {
        [ 'The drone of aircraft engines can be heard overhead. ',
          'From somewhere across the city comes the wail of a distant siren. ',
          'From somewhere to the ' + dirn + ' comes the bark of anti-aircraft fire. ',
          'Off to the ' + dirn + ' you hear the blast of a whistle and the sound
            of running feet. ',
          'A fire engine races down a street somewhere to the ' + dirn + '. ',
          'There\'s a sudden explosion somewhere off to the ' + dirn + ', as
            another bomb finds a target. '
        ]
        dirn = (rand('north' , 'south' , 'east' , 'west'))
    }
;
;
```

We override the `discover()` method of the bomb to set up the new `SenseFuse`, in such a way that the player character must be in a position to see it for the messages in `explode()` to be displayed to the player. In this case we could have achieved much the same effect by incorporating the message that's displayed into the group of statements governed by the `if(gPlayerChar.isIn(location))` statement, but that would not always be so convenient, and we're trying to illustrate a `SenseFuse`!

Later on we'll make this bomb a bit more interesting, for example by adding a [Noise](#) object to make it tick. In the meantime note the use of the `specialDesc` property on the rubble; we override `useSpecialDesc` so that this `specialDesc` is displayed only after the bomb has gone off, and we set `specialDescOrder` to 70 (lower than the default of 100) so that the description of the new bomb crater etc. will come earlier in the listing of the room contents than other `specialDescs` and `initDescs`.

10.4. SenseDaemon

A SenseDaemon, like a [SenseFuse](#), is for use when you want players to see its output only when a certain object can be sensed.

The way to set up a SenseDaemon is very like that of setting up an ordinary Daemon, or a SenseFuse, namely:

```
new SenseDaemon(obj, &prop, interval, source, sense);
```

As with the standard Daemon, this sets up a daemon that will call the method `obj.prop` every *interval* turns, where *interval* must be at least 1 (one). If *interval* is 1, `obj.prop` will execute each turn, starting with the current turn. If *interval* is 2, `obj.prop` will execute every other turn, starting with the next turn. In general, if *interval* is *n*, `obj.prop` will execute every *n* turns, starting in *n-1* turns time. Note once again that the second argument to the new Daemon call must be a property (or method) *pointer*, hence the `&prop` syntax. The difference is that the player will only see any messages displayed by `obj.prop` if, at that point in time, the player character can sense the *source* object (which in practice may often be the same as the `obj` object, but need not be) using the sense *sense* (which could be sight, sound, smell or touch - most likely one of the first two).

We'll create an example of a SenseDaemon when we come to [CyclicEventList](#).

10.5. PromptDaemon

A PromptDaemon is a special kind of daemon that runs once each turn, just before the command prompt is displayed. This may be useful, for example, where you want to check each turn whether some condition has become true and take appropriate action if so. Since the PromptDaemon runs every turn, there is no need to specify its frequency, so you can set one up simply with the command:

```
newPromptDaemon(obj, &prop);
```

Which will call `obj.prop` each turn, just before the command prompt is displayed. Again it may be useful to make a note of a reference to the PromptDaemon so that it can be removed from the list of active events once its job is done, e.g.

```
daemonID = newPromptDaemon(obj, &prop);
```

Then, when you've finished with the promptDaemon you can simply call:

```
daemonID.removeEvent();
```

An example of the possible use of a PromptDaemon is given later in connexion with a [bomb](#).

10.6. OneTimePromptDaemon

A OneTimePromptDaemon is a special kind of PromptDaemon that automatically deactivates itself after its first invocation, thereby ensuring that it is executed only once. A one-time-only prompt daemon is a regular command prompt daemon, except that it fires only once. After it fires once, the daemon automatically deactivates itself, so that it won't fire again.

Prompt daemons are occasionally useful for non-recurring processing, when you want to defer some bit of code until a "safe" time between turns. In these cases, the regular PromptDaemon is inconvenient to use because it automatically recurs. This subclass is handy for these cases, since it lets you schedule some bit of processing for a single deferred execution.

One special situation where one-time prompt daemons can be handy is in triggering conversational events - such as initiating a conversation - at the very beginning of the game. Initiating a conversation can only be done from within an action context, but no action context is in effect during the game's initialization. An easy way to deal with this is to create a one-time prompt daemon during initialization, and then trigger the event from the daemon's callback method. The prompt daemon will set up a daemon action environment just before the first command prompt is displayed, at which point the callback will be able to trigger the event as though it were in ordinary action handler code. We can't

TADS 3 Tour Guide

use a regular Fuse or Daemon for this, since a regular Fuse or Daemon will only fire at the *end* of the player's turn, and in the case just described, we need something that fires just *before* the first command prompt appears.

So, for example, to have an NPC (let's call her Sarah) initiate a conversation, posing a question to the player character (such as "What are you doing here?") just before the very first turn, we might could do something like this:

```
OneTimePromptDaemon, InitObject
  execute() { construct(self, &beforePrompt); }
  beforePrompt()
  {
    sarah.initiateConversation(sarahTalking, 'query-presence');
  }
;
```

Don't worry about how [iniateConversation](#) works for now, we'll be going into that later. For now it suffices to know that this is the statement we need to execute to get Sarah chatting just before the first turn, so that the player's first command can be a require to her question. The point of OneTimePromptDaemon is to provide us somewhere where we can put this statement and have it execute when we need it.

Note that there's nothing magical about the name beforePrompt() here; we could have called it anything we liked, although beforePrompt() is at least descriptive of when the method is invoked, and might be useful if we wanted to define a custom subclass for this kind of situation:

```
class FirstTurnPromptDaemon: OneTimePromptDaemon, InitObject
  execute() { construct(self, &beforePrompt); }
  beforePrompt() {}
;
```

Rather more noteworthy here is the combination of OneTimePromptDaemon with [InitObject](#), which we'll discuss next.

11. ModuleExecObjects

11.1. ModuleExecObject

ModuleExecObjects are a little like Fuses and Daemons in that they allow code to be executed at a particular point, although a ModuleExecObject is not the really same thing as a Fuse or Daemon. Instead, ModuleExecObject is an abstract base class for various classes that provide modular execution hooks. This class and its subclasses are mix-in classes - they can be multiply inherited by any object (as long as it's not already some other kind of module execution object).

The point of the Module Execution Object and its subclasses is to allow libraries and user code to define execution hooks, without having to worry about what other libraries and user code bits are defining the same hook. When we need to execute a hook defined via this object, we iterate over all of the instances of the appropriate subclass and invoke its `execute()` method.

By default, the order of execution is arbitrary. In some cases, though, dependencies will exist, so that one object cannot be invoked until another object has already been invoked. In these cases, you must set the `execBeforeMe` property to contain a list of the objects whose `execute()` methods must be invoked before this object's `execute()` method is invoked. The library will check this list before calling `execute()` on this object, and ensure that each object in the list has been invoked before calling this object's `execute()`. Similarly, you can use the `execAfterMe` property to contain a list of all the ModuleExecObjects that the current object must execute before.

11.2. InitObject

An InitObject is an object that contains an `execute()` method that is executed at the start of the game, before the first command prompt occurs. In a particular game much the same effect could be achieved by putting the code in `gameMain.showIntro()`, but there are occasions when you might prefer to use InitObject for a particular task, for example:

- Starting up a Daemon or Fuse, when the InitObject can conveniently be mixed-in with the Daemon or Fuse to form a single object (as in the [OneTimePromptDaemon](#) example above).
- Writing code for a library extension, or code that you want to be reusable between games, for which isolating it in a separate object will be far more convenient than placing it in a game-specific `gameMain.showIntro()` method.
- Writing code for a custom class or object that you want to have initialize itself at startup.

Some control is possible over the order of execution of InitObjects by stipulating a list of InitObjects that must be executed before the current one in the `execBeforeMe` property, and a list of InitObjects that must be executed after the current one in the `execAfterMe` property (this mechanism is common to all ModuleExecObjects).

Note that in many cases, however, it may be better to use a [PreinitObject](#) for most of these purposes. One case where you *must* use an InitObject rather than a PreinitObject is when you want the object to set up a Fuse or Daemon. Your code will probably compile if you do this in a PreinitObject, but you'll find that the Fuse or Daemon is not actually set up when the game runs. Another case where you would need to use an InitObject rather than a PreinitObject is where you want its `execute` method to randomize something at the beginning of the game, e.g.:

```
InitObject
execute()
{
    gameMain.villain = rand(moriarty, darthVader, presidentClark, caligula);
    gameMain.villain.moveIntoForTravel(startRoom);
}
;
```

Although in this example there'd probably be little reason for not putting such code in `gameMain.showIntro()`. This might be different if we had a class of objects that we wanted to set themselves up in this sort of manner at the start of the game, e.g.:

TADS 3 Tour Guide

```
class PredatoryMale: Person, InitObject
    mainTargetOfDesire = nil
    execute()
    {
        mainTargetOfDesire = rand(jane, jill, sandra, mary, virginia);
    }
;
```

The fact that this might allow several predatory males to share the same main target of desire might not matter in the least in this imaginary game (although in this case we could have overridden `initializeActor` to achieve the same result).

11.3. PreinitObject

A `PreinitObject` works in exactly the same way as an `InitObject` with one important difference, its `execute` method is executed at the preinitialization stage, not at game startup. Preinitialization is carried out by the *compiler* before the game image is written rather than by the *interpreter* when the game starts, and so is useful for any game initialization code that will always have the same results (since, having been already carried out at compilation, it doesn't need to be carried out at game startup when the player might otherwise notice a delay in the game starting).

For example, suppose that instead of randomizing which woman each of our predatory males fancies at the start of the game, we want to define this for ourselves (so that it never changes from game to game), but that for ease of computation at some point in our game it's convenient for each of the fanciable woman to maintain a list of the men who are after them. We might define:

```
class FanciableWoman: Person
    fanciedBy = []
;

class PredatoryMale: Person, PreinitObject
    mainTargetOfDesire = nil
    execute()
    {
        if(mainTargetOfDesire != nil)
            mainTargetOfDesire.fanciedBy += self;
    }
;
```

Or indeed, in the interests of gender equality, we might have:

```
class AmorousPerson: Person, PreinitObject
    fanciedBy = []
    mainTargetOfDesire = nil
    execute()
    {
        if(mainTargetOfDesire != nil)
            mainTargetOfDesire.fanciedBy += self;
    }
;
```

Which would allow us to set up whatever tangled web of relationships or would-be relationships we wish, even including:

```
narcissus: AmorousPerson 'narcissus' 'Narcissus'
    isHim = true
    mainObjectOfDesire = self
;
```

An alternative (and not uncommon) way to use a `PreinitObject` to achieve roughly the same result where we want every `Person` in the game to be potentially involved in amorous activities and we don't want to define a new class for it would be the following:

TADS 3 Tour Guide

```
modify Person
    fanciedBy = []
    mainTargetOfDesire = nil
;

PreinitObject
    execute()
    {
        for(local obj = firstObj(Person); obj != nil; obj = nextObj(obj, Person));
        {
            if(obj.mainTargetOfDesire != nil)
                obj.mainTargetOfDesire.fanciedBy += obj;
        }
    }
;
```

11.4. PreSaveObject

PreSaveObject - every instance of this class is notified, via its execute() method, just before we save the game. This provides a convenient way to make something happen just before our game is saved. The need for this may not arise very often, but if it should arise it's probably more convenient to define a PreSaveObject than to tinker about trying to modify SaveAction or something it calls.

For example, an author really determined to earn the wrath and indignation of his players might write:

```
PreSaveObject
    execute()
    {
        "You pathetically pusillanimous poltroon! <i>Real</i> adventures don't need to save!
        To penalize you for this cowardly inanity ten thousand points will be instantly
        deducted from your score.<.p>";

        addToScore(-10000, 'saving the game');
    }
;
```

Of course, we don't recommend you follow such an author's example!

11.5. PostRestoreObject

A PostRestoreObject is similar to a [PreSaveObject](#), except that instead of its execute method being invoked just before saving, it is invoked just after restoring.

For example, if our imaginary mad IF author (he'd have to be mad, wouldn't he) wanted to inspire not only anger but hatred and loathing, he could add the following:

```
PostRestoreObject()
    execute()
    {
        "You ridiculous wretch! To be restoring this game you must have saved it,
        and you <i>know</i> how much I hate that! I suppose you think that way you
        can escape the points penalty for saving. Well, you can't -- I'm deducting
        another ten thousand points for restoring, and IT SERVES YOU RIGHT!!!!<.p>";

        addToScore(-10000, 'restoring the game');
    }
;
```

It goes without saying, of course, that the good reader of this guide would never use PostRestoreObject for so nefarious a purpose, but it could happen that there was some legitimate housekeeping activity we needed to carry out just after a restore (for example, if we were trying to keep track of how long the player had been playing our game for).

11.6. PostUndoObject

PostUndoObject - every instance of this class is notified, via its execute() method, immediately after we perform an 'undo' command.

You can probably guess how our manic player-hating author might put this to evil use:

```
PostUndoObject
execute()
{
    "If there's one thing I hate more than players feeble enough to save and restore,
    it's idiots who ruin the amazing gaming experience I've devised for them by
    resorting to UNDO. The penalty of <i>fifty</i> thousand points you're about to
    suffer is thus richly deserved.\b
    By the way, this makes the game unwinnable - but hey, only lousy LOSERS like
    you use UNDO.<.p>";

    addToScore(-50000, 'using undo');
}
;
```

Hopefully if you do find yourself using a PostUndoObject it will be for a more legitimate purpose than this, though it's probably not something you'll need to use very often, if at all. A more legitimate use might be if, for some research purpose, you wanted to keep track of how often players used UNDO during a session with your game, then you might do something like this:

```
transient statisticsObj: object
    undoCount = 0
    saveCount = 0
    restoreCount = 0
;

PostUndoObject
execute()
{
    statisticsObj.undoCount += 1;
}
;
```

Because statisticsObj has been declared as **transient** (for fuller details of which see the Object Definitions section of the language documentation) its properties will be preserved across operations such as UNDO, RESTART, SAVE and RESTORE, and this should work (although since a transient object is thus not itself saved or restored, its properties cannot be preserved across different game sessions, hence the seemingly odd name 'transient' for this type of object).

11.7. PreRestartObject

PreRestartObject is the last member of this set. Every instance of this class is notified, via its execute() method, just before we restart the game (with a RESTART command, for example).

Our totally insane implementor would probably use it like this:

```
PreRestartObject
execute()
{
    "NO!! You pathetic worm! I have filled this game with a myriad unmappable
    mazes, an infinity of instadeath rooms, a glut of guess-the-verb puzzles,
    an unbounded cornucopia of unimplemented objects, and you -- you, you miserable
    wretch -- you want to back out of all this by RESTARTING! Well, I'm not
    having it. You'll just have to carry on. So there! ";

    exit;
}
;
```

TADS 3 Tour Guide

At this point you may be feeling thoroughly grateful that there's no PreQuitObject. Unfortunately our Insane Implementor could always resort to:

```
modify QuitAction
    execSystemAction()
    {
        "What? You want to quit my masterpiece? No way! What kind of cretin are
        you, anyway?";
    }
;
```

But please don't try this at home!

And now, after that little diversion, let's get back to the Quest of the Golden Banana.

12. Pushing Things Around

12.1. TravelPushable

A `TravelPushable` is an object that can't be picked up but can be pushed from one location to another (by the player issuing commands such as `PUSH WHEELBARROW NORTH` or `PUSH BARROW INTO SHIP`). As a simple example we can implement a wheelbarrow that would be useful for moving the gold tablet around:

```
wheelBarrow : TravelPushable, Container 'old tin wheel wheelbarrow/barrow'
  'wheelbarrow' @graveyard
  "It's an old tin wheelbarrow, a bit battered, but seemingly still
    in serviceable condition. "
  initSpecialDesc = "An old tin wheelbarrow lies forgotten in one corner of the graveyard. "
  specialDesc = "The old tin wheelbarrow rests here. "
  cannotTakeMsg = 'The wheelbarrow is a bit too heavy and cumbersome to carry around.
    Pushing it would probably prove more productive. '
;
```

Since `TravelPushable` inherits from `Immovable`, it's a good idea to give the wheelbarrow a `specialDesc` as well as an `initSpecialDesc` to make sure that it gets listed in room descriptions. Although the default message you get when you try to take a `TravelPushable` is okay, we customize it here to say the same thing but in a manner slightly more tailored to the specific object; we do this by overriding `cannotTakeMsg` with a single-quoted string (it must contain either a single-quoted string or a property pointer).

This wheelbarrow will work fine as it is, and will certainly help with carting the gold tablet around. One refinement we could add is to improve on the "You push the wheelbarrow into the area" that appears each time the barrow is pushed somewhere. We can take advantage of the fact that we've defined a `destName` on every location to override `describeMovePushable` with something a bit more specific:

```
modify TravelPushable
  describeMovePushable(traveler, connector)
  {
    if (gActor.isPlayerChar)
      "You push <<theName>> into <<location.destName>>. ";
  }
;
```

For a more complex example, let's start filling in some of the detail on the south side of the lake. The tunnel south from the shore will soon come to a deep uncrossable chasm. However, a stone monolith waits on the shore; if the player pushes it into the chasm, the monolith then forms a bridge that can be crossed. The main trick here is to make the chasm a room in its own right, and, recognizing that a `Room` is also a `TravelConnector`, override its `canTravelerPass` and `explainTravelBarrier` methods to prevent travel unless either the monolith is already in the chasm or it is in the process of being pushed into the chasm:

```
southShore : Room 'Rocky shore' 'the rocky shore'
  "The rocky shore looks so barren here as to be scarcely worth visiting, apart
    from a narrow tunnel leading off to the south"
  finalDesc = ". " // for the custom finalDesc property see our inRoomDesc modification
  south = narrowTunnel
;

+ monolith : TravelPushable 'large black monolith' 'black monolith'
  "Deep black in colour, it's a smooth black oblong about six feet by three, and
    about six inches thick. <<isIn(deepChasm) ? 'Currently, it\'s wedged in the
    chasm, forming a precarious bridge. ' : nil>>"
  initSpecialDesc = ""
  specialDesc = "<<isIn(deepChasm) || travelInProgress ? '
    : 'The black monolith stands on the ground. '>>"
  inRoomDesc() { if(!moved) ", and a black monolith projecting out of the rocks"; }
  cannotTakeMsg = 'There\'s no way anyone\'s going to lift that great block of stone. '
  dobjFor(PushTravel)
  {
    verify()
    {
      if(isIn(deepChasm))
```

TADS 3 Tour Guide

```
{
    illogicalNow ('For one thing you\'re standing on the monolith, and
        for another it\'s wedged firmly (you hope) in the chasm. ');
}
else
    inherited;
}
action
{
    travelInProgress = true;
    inherited;
    travelInProgress = nil;
}
}
travelInProgress = nil
describeMovePushable(traveler, connector)
{
    if (location == deepChasm)
    {
        "The monolith has toppled into the chasm, forming a precarious bridge of sorts,
            on which you're now standing. ";
        setSuperclassList([Floor]);
        deepChasm.roomParts += self;
    }
    else
        inherited(traveler, connector);
}
beforeMovePushable(traveler, connector, dest)
{
    if(connector == deepChasm)
        "You push the monolith towards the edge of the chasm; as it reaches
            the edge it begins to topple.\b";
    else
        "The monolith is <i>very</i> heavy, but with a supreme effort
            than nearly gives you three hernias and four burst blood
            vessels you manage to start pushing it. ";
}
dobjFor(SitOn)
{
    preCond = inherited + actorDirectlyInRoom
    verify()
    {
        if(ofKind(Floor) && gActor.isIn(deepChasm))
            logicalRank(140, 'most likely floor' );
        else if(ofKind(Floor))
            nonObvious;
        else
            inherited;
    }
}
dobjFor(LieOn)
preCond = inherited + actorDirectlyInRoom
{
    verify() { if(ofKind(Floor)) verifyDobjSitOn; else inherited; }
}
;

narrowTunnel : DarkRoom 'Narrow Tunnel' 'the narrow tunnel'
    "This short section of tunnel leads south from the rocky shore, but then
        comes to an abrupt end at the edge of a deep chasm. Another tunnel continues
        south from the ledge on the far side of the chasm. "
    north = southShore
    south = deepChasm
;

MultiLoc, Enterable ->deepChasm 'deep chasm' 'deep chasm'
    "<<deepChasm.desc>>"
    locationList = [narrowTunnel, chasmLedge]
;

deepChasm : DarkRoom 'Deep Chasm' 'the deep chasm'
    "The chasm is not something you want to look down if you suffer from
        vertigo; but a long way below, almost too far down to see, runs a narrow, inky river.
```

TADS 3 Tour Guide

```
The chasm is about six feet wide, too far to jump<<monolith.isIn(self) ? ', but a
stone monolith forms a bridge of sorts across it' : nil>>. "
north = narrowTunnel
south = chasmLedge
canTravelerPass(traveler)
{return monolith.isIn(self) || monolith.travelInProgress; }
explainTravelBarrier(traveler)
{
    "The chasm is too wide to jump over, and you certainly don't
    want to fall into it. ";
}
cannotGoThatWayMsg = 'Stepping off the monolith into the chasm would mean
falling to almost certain death. '
roomParts = [defaultCeiling]
lookAroundWithinName(actor, illum)
{
    inherited(actor, illum);
    if (actor.posture == standing)
        " (standing on <<monolith.theName>>)" ;
}
;

+ inkyRiver : Distant 'narrow inky black river/water' 'river'
"The narrow river runs along the bottom of the chasm about a hundred feet or so below;
its water looks inky black in the near darkness down there. "
;

chasmLedge : DarkRoom 'Ledge of Chasm' 'the ledge of the chasm'
"A deep, wide chasm splits the ground immediately to the north of this
narrow ledge, while a dark tunnel runs south. Another tunnel can be
seen leading north from the far side of the chasm. "
north = deepChasm
;
```

One trick here is to override `deepChasm.lookAroundWithinName` so that it adds "(standing on the black monolith)" to the name of the room when the player character is indeed standing (the inherited behaviour will deal with sitting and lying).

The other trick here is to make the monolith transform itself into a Floor and add itself to the chasm's `roomParts` once it arrives in the chasm, so that we get the right responses if the player SITS or LIES there. Also, since the `defaultFloors` of the adjacent rooms will also be in scope thanks to the [DistanceConnector](#) we'll be adding later, we need to make the monolith the most likely item to sit or lie on when we're directly on top of it, but not when we're in one of the adjacent locations. Conversely, we add `actorDirectlyInRoom` to the preconditions for sitting or lying on the monolith to prevent the otherwise rather odd behaviour that will occur (once the `DistanceConnector` is in place) when the player character attempts to sit or lie on the monolith while he's in an adjacent location.

12.2. PushTravelBarrier

A `PushTravelBarrier` is a special kind of [TravelBarrier](#) that can be used (by attaching it to the `travelBarrier` property of a `TravelConnector`) to block (or selectively block) pushing a `TravelPushable` via this connector. By default a `PushTravelBarrier` blocks all `TravelPushables`, but this can be changed by overriding its `canTravelerPass` method, or, perhaps more simply, its `canPushedObjectPass(obj)` method, which `canTravelerPass` calls. You can also override the `explainTravelBarrier` method to explain why pushing an object this way isn't allowed.

For example, we might well want to prevent pushing any `TravelPushables` up and down stairs or ladders. To do this, we can simply define an appropriate `PushTravelBarrier` object and modify the `Stairway` class to make use of it:

```
modify Stairway
    travelBarrier = [stairBarrier]
;

stairBarrier : PushTravelBarrier
    explainTravelBarrier(traveler)
    {
```

TADS 3 Tour Guide

```
        local obj = traveler.obj_;
        gMessageParams(obj);
        reportFailure('{The obj/he} can\'t negotiate ladders and stairs. ');
    }
;

```

Although we allow the monolith to be pushed down the tunnel, we may feel that it shouldn't be possible to push it aboard the ship. This time we can create a selective PushTravelBarrier that just blocks the monolith:

```
monolithBarrier : PushTravelBarrier
    canPushedObjectPass(obj) { return obj != monolith; }
    explainTravelBarrier(traveler)
    {
        "The monolith is far too large and unwieldy to be pushed there. ";
    }
;

```

Then all we have to do is attach this barrier to the portDeck of the ship (which is where anything entering the ship will arrive):

```
portDeck : Deck 'Port Deck' 'the main deck'
    "This part of the main deck is on the port side of the ship, close to the shore. The
    deck continues to fore, aft and starboard, and a tall mast towers up from
    the middle of the main deck. "
    fore = foreDeck
    aft = quarterDeck
    starboard = starboardDeck
    out = (ship.location)
    up = mast
    travelBarrier = [monolithBarrier]
;

```

Once again, this works because a Room (from which the custom Deck class inherits) inherits from TravelConnector.

13. Intangibles & Senses

13.1. Intangibles - Overview

Intangibles in the TADS 3 library constitute things that have some kind of presence but are not physical objects, such as light, smoke, smells and sounds. The class hierarchy for Intangible is:

```

Intangible
  DistanceConnector

  SensoryEmanation
    Noise
      SimpleNoise
    Odor
      SimpleOdor

  Vaporous

SenseConnector
  DistanceConnector

```

While we're looking at intangibles and sensory things, we'll also cover:

```

SensoryEvent
  SoundEvent

SoundObserver

```

13.2. Intangible

An Intangible object (as opposed to an object derived from some of Intangible's subclasses) is one that has no sensory presence whatsoever, which means that the player can never refer to it, even with a bare EXAMINE command. That means it is really only useful for abstract objects that the player will never interact with directly (as a mix-in class with a SenseConnector, perhaps, as in the [DistanceConnector](#)). To represent intangible but sensible objects such as a ray of light, you are better off using the [Vaporous](#) class than trying to tweak Intangible.

13.3. DistanceConnector

Let's go back to the chasm we created between the narrow tunnel from the south shore and the opposite ledge. Since the chasm is only six feet wide, you would have thought that once you had pushed the monolith into it to form the bridge, you'd be able to see the monolith from either side of the chasm (as well as the chasm itself). Furthermore, with this setup, you'd expect that if a torch/flashlight were left on one side of the chasm you'd still have light enough to cross the chasm by. In fact, if you try putting this to the test, the transcript you'll see is something like this:

```

>push monolith south
Deep Chasm (standing on the monolith)

```

The chasm is not something you want to look down if you suffer from vertigo; the bottom is far out of sight in the impenetrable blackness below. It is about six feet wide, too far to jump.

The monolith topples into the chasm, forming a precarious bridge of sorts, onto which you step.

```

>s
Ledge of Chasm

```

A deep, wide chasm splits the ground immediately to the north of this narrow ledge, while a dark tunnel runs south. Another tunnel can be seen leading north from the far side of the chasm.

TADS 3 Tour Guide

>x monolith

You see no monolith here.

>drop torch

Dropped.

>n

In the dark

It's pitch black.

What happens, of course, is that the game treats the two sides of the chasm and the chasm itself as three entirely separate locations; even if in our own mind's eye they are closely related in the sense of being in close physical proximity and visually open to one another, there's no way the game can read our mind and know this. There is, however, a way we can tell the game that we want locations to be connected in this way, and that is to use a `DistanceConnector`. This is basically a way of linking separate locations by sight without making them into a single location, so that we can break up a large location such as a big hall, a large town square, or (as in this case) a chasm and its two sides into separate rooms while maintaining visual contact between them.

To use a `DistanceConnector` is extremely easy. You just need the class name followed by a list of the locations being connected. In the present case we just need to define:

```
DistanceConnector [narrowTunnel, deepChasm, chasmLedge] ;
```

You should now be able to push the monolith into the chasm, carry on to the other side, `EXAMINE` the monolith from there, drop the torch there and walk back across the chasm still by the light of the torch. When you `EXAMINE` the monolith, though, you may be told "It's too far away to make out any detail", which doesn't seem too reasonable given that it's a large block of stone right at your feet. The way to deal with this is to add the following line to the definition of the monolith:

```
sightSize = large
```

By default `sightSize` (and `smellSize`, `soundSize` and `touchSize`) are all medium, which means that the object can be sensed at a distance but not well enough to make out any detail. Setting `sightSize = small` would mean that we could not even see the object at a distance, while setting it to `large` means that we can not only see it but make out the details (i.e. see the normal `desc` property in response to an `EXAMINE` command).

Once we can see things at a distance we may be faced with another problem: what can be seen from a distance may not be the same as what can be seen close-to. Thus the way we want objects to be described from a distance may be different from their standard (close-up) description. If we use an `initSpecialDesc` or `specialDesc` we can also define the corresponding properties `remoteInitSpecialDesc` and `remoteSpecialDesc` to use when the object is viewed from another location, for example:

```
museumLeaflet : Readable 'small crumpled piece yellow leaflet/paper' 'yellow leaflet'
  @chasmLedge
  "It seems to be a leaflet advertising the Eerhtstad Caves Museum of
  Curious Antiquities. "
  readDesc = "Amongst such oddities as the Amber Amulet of Amazement, the
  Green Gargoyle of Gloom, the Lost Crown of King Peregrine the Pipsqueak
  and the Invisible Mantle of the Naked Emperor, your eye is caught by
  an advertisement for the star exhibit: the Golden Banana of Discord.
  The reverse of side of the leaflet proclaims, <q>Our most wanted
  acquisition of the month is the legendary Great Purple Carbuncle of
  King Solomon; if discovered, please bring to the curator who will
  not only receive it with <i>great</i> gratitude but ensure that a small
  brass plaque is inscribed to your everlasting honour!</q><.p>"
  initSpecialDesc = "A small yellow leaflet lies on the ground. "
  remoteInitSpecialDesc(actor)
  {
    "On the <<actor.isIn(deepChasm) ? 'southern' : 'far'>>
    ledge of the chasm lies a crumpled piece of yellow paper. ";
  }
;
```

Note that `remoteInitSpecialDesc` takes a single parameter, `actor`. This normally represents the actor from whose point of view the object is to be described. If this affects the remote description at all, it will normally depend on the

TADS 3 Tour Guide

location of the actor; thus we can use it here to vary the description of the location of the leaflet depending on the location of the player character when it's described (if we wanted to, we could also vary the description of the leaflet itself, perhaps making it slightly more detailed from the closer point of view).

There's a further refinement we can make here: if you push the monolith into the chasm, walk south to the ledge, take the leaflet and drop it, then return to the monolith, the room description will now state:

In the ledge of chasm, you see a yellow leaflet.

This shows that the leaflet is still visible, but that it's in the 'ledge of chasm' room, not the current location. However, 'In the ledge of chasm' is not the most felicitous way to describe the position of the leaflet, so we may want to customise it. We can do this by overriding the `inRoomName(pov)` method of `chasmLedge` (where `pov` has the same meaning as before), for example:

```
chasmLedge : DarkRoom 'Ledge of Chasm' 'the ledge of the chasm'
  "A deep, wide chasm splits the ground immediately to the north of this
  narrow ledge, while a dark tunnel runs south. Another tunnel can be
  seen leading north from the far side of the chasm. "
  north = deepChasm
  inRoomName(pov)
  {
    return 'on the ' + (pov.isIn(deepChasm) ? 'south' : 'far') + ' ledge of the chasm';
  }
;
```

13.4. Occluder

While a `DistanceConnector` allows you to see more objects from a particular location than you normally would (by letting you see what's in neighbouring locations) an `Occluder` can remove objects from visibility (or the scope of any other sense).

The reason you may want to do this is that what is visible can depend on where you're looking from. For example, suppose you have a window looking into a living-room. You could implement the window as a [SenseConnector](#) to make it possible for the game to list the contents of the room when the player character is on the outside looking in. But suppose there's a bookcase on the same wall as the window; then the player character probably ought not to be able to see either the bookcase or anything that's in the bookcase from outside the window, although the bookcase and its contents are perfectly visible from anywhere within the living-room. To achieve this, you'd use an `Occluder` to remove the bookcase and its contents from sensory scope if the point of view was in the location on the outside of the window, defining what you want occluded in the `Occluder's occludeObj(obj, sense, pov)` method.

To make an `Occluder` occlude an object for a particular combination of object (the `obj` parameter), sense, and point of view (the `pov` parameter), you just make the `occludeObj` method return true for that combination (and nil otherwise). In the bookcase example the `occludeObj` method might thus look something like:

```
occludeObj(obj, sense, pov)
{
  return (obj.isIn(bookcase) || obj == bookcase) && pov.isIn(outsideWindow);
}
```

(Where `outsideWindow` is the name of the location on the outside of the window). Note that here we have made no use of the `sense` parameter, since we want occlusion to occur (or not occur) for all senses; this is probably the most common case, but there may be exceptions (for example a loudly ticking clock placed on the bookcase might conceivably be audible from outside the window), in which case you'd also need to test for the `sense` parameter.

Since we haven't got a convenient window and bookcase to hand in the *Quest of the Golden Banana*, we'll use a different example here. You'll recall that the description of the `deepChasm` room mentions a narrow river visible a hundred feet below at the bottom of the chasm, and that we added a `Distant` object to represent this river in case the player tries to refer to it (see the [TravelPushable](#) section above). Now that we've added a `DistanceConnector` linking `deepChasm` to the rooms either side of it (`chasmLedge` and `narrowTunnel`), the `inkyRiver` object will be in scope from those two locations too. However, since the chasm is so deep the river at its bottom should only be visible when the player character is actually in `deepChasm` (on the monolith, looking straight down). If he is one of the locations neighbouring the chasm, the chasm walls will occlude the river from view.

TADS 3 Tour Guide

Occluder is a mix-in class, which means that we need to mix it in with another class. We don't need to create an object specially for the purpose, however; in this case the DistanceConnector we've just created will prove entirely suitable. We can now modify it to incorporate the Occluder:

```
Occluder, DistanceConnector [narrowTunnel, deepChasm, chasmLedge]
  occludeObj(obj, sense, pov)
  {
    return obj == inkyRiver && !pov.isIn(deepChasm);
  }
;
```

If you now recompile and run the game you should find that the river is now visible only from the chasm.

Note: although we incorporated the Occluder into the DistanceConnector for convenience, the action of the Occluder is not restricted to that DistanceConnector. If, at a later stage, we were to add an overhanging ledge some way above the chasm, and then add another, completely separate DistanceConnector, linking the overhanging ledge and the chasm, the river would not be visible from the overhanging ledge, since the Occluder we've already created would still occlude it (unless we modified its occludeObj method to do something different). This behaviour of Occluder is actually pretty useful, since it enables you to define all the occlusion rules for a given location in one place, but it can be a little confusing if you're not expecting it.

For this reason, if you have a location joined to several other locations by different DistanceConnectors (or SenseConnectors), you may find it more intuitive to create a separate object, such as a SecretFixture, to act as the Occluder. For example, instead of adding the Occluder to the DistanceConnector as in the example above, we could have added a SecretFixture to the deepChasm room (say, immediately after the definition of inkyRiver):

```
+ Occluder, SecretFixture
  occludeObj(obj, sense, pov)
  {
    return obj == inkyRiver && !pov.isIn(deepChasm);
  }
;
```

And it would have worked in exactly the same way.

Note there is absolutely no need to do it this way in such a case; you may simply find it clearer to do so. In particular, if you have a complex set of interconnections via DistantConnectors with different objects occluded from different points of view, you may find it easier to set up an Occluder, SecretFixture in each of the linked locations, and have each one start by *not* occluding anything when the pov is in the same location, e.g.:

```
+ Occluder, SecretFixture
  occludeObj(obj, sense, pov)
  {
    if(pov.isIn(roomLocation))
      return nil;

    if(obj.ofKind(Distant))
      return true;

    if(pov.isIn(garden))
      return (obj == sideboard || obj.isIn(sideboard));

    ...
  }
;
```

In this example we start by ensuring that nothing is Occluded if the point of view is in the same room as the Occluder - this can be useful in complex cases to ensure we don't end up occluding something by accident. We then occlude all Distant objects (we might want to do this because an object implemented as Distant in one location might be implemented rather differently in a neighbouring location); note that this will *not* occlude Distant objects when the pov is in the same location, since in this case occludeObj will already have return nil before getting to this test. Next we make the sideboard and its contents invisible from the garden. We could then go on to add any further occlusion rules we wanted.

As from TADS 3.0.9 it is possible to let individual objects decide whether they are occluded by a given occluder. By default the TADS 3.0.9 library defined Occluder.occludeObj thus:

TADS 3 Tour Guide

```
occludeObj(obj, sense, pov)
{
    /* by default, simply ask the object what it thinks */
    return obj.isOccludedBy(self, sense, pov);
}
```

While Thing.isOccludedBy() is defined in the library as:

```
isOccludedBy(occluder, sense, pov) { return nil; }
```

Thus, for example, if there was a window through which you could see into the sitting room from the garden, but you couldn't see the bookcase against the wall and the mirror hanging on the wall, instead of trying to write the appropriate occludeObj() routine on the Occluder, you could write isOccludedBy routines on the mirror and bookcase:

```
mirror: Fixture 'plain square mirror' 'mirror'
    "It's a plain square mirror in which you can see your reflection quite clearly. "
    isOccludedBy(occluder, sense, pov)
    {
        return occluder == window && pov.isIn(garden);
    }
;
```

And likewise for the bookcase. Whether you find it easier to write the occlusion rules on the Occluder or on individual objects depends partly on the situation and partly on personal taste. As a general rule, the more complicated the situation, and the more different cases you need to take into account for the more objects, the more likely an occludeObj() routine written on the Occluder is likely to be bug-ridden, and the safer it might be to write your occlusion rules on individual objects. On the other hand, where there's a simple occlusion rule (e.g. we don't want any objects of class Distant located in the garden to be visible when looking through the window from inside the house), the simpler it might be to write the rules in the occludeObj() routine of the Occluder. It is, of course, possible to combine both approaches:

```
window: Occluder, Fixture 'window' 'window' @sittingRoom
...
occludeObj(obj, sense, pov)
{
    return (obj.ofKind(Distant) && obj.isIn(garden) && pov.isIn(sittingRoom))
        || inherited(obj, sense, pov);
}
;
```

13.5. Vaporous

Vaporous is a good class to use for something you can see, and maybe smell and hear, but that is not fully tangible, such as a ray of light, a flame, or smoke. To set up an example, let's start creating some locations on the east side of the lake:

```
eastShore : Room 'Stone Jetty' 'the stone jetty'
    "This bleak stone jetty is little more than a narrow corridor between the lake to
    the west and the rough cave wall to the east. A broad flight of stone steps leads
    down to the south, while a much narrower flight leads up to the north. "
    south = eastShoreDown
    down asExit(south)
;

+ eastShoreDown : StairwayDown 'broad flight stone steps' 'broad stone steps'
    "The broad stone steps looks fatally inviting, an easy walk down into the
    bowels of the earth. "
    isPlural = true
;
```

TADS 3 Tour Guide

```
hellVestibule : Room 'Vestibule of Hell Fire Cavern' 'the vestibule'
    "The broad stone steps leading up to the north come to an end in this small, hot,
    rough round cave that seems to form the vestibule to what lies beyond the
    uninviting entrance to the east, through which comes a dull red glow. A
    sign next to this entrance declares it to be the entrance to Hell Fire Cavern. "
    north = hellVestibuleUp
    up asExit(north)
;

+ hellVestibuleUp : StairwayUp ->eastShoreDown 'broad stone steps' 'broad stone steps'
    "The steps back up to the jetty look long, rough and wearisome. "
    isPlural = true
;

+ Readable, Decoration 'sign' 'sign'
    "The sign declares:\n
    <b><FONT COLOR=RED>HELL FIRE CAVERN</FONT></b>\n
    ADMISSION ABSOLUTELY FREE\n
    (getting out alive not guaranteed)\n"
;
```

The description of hellVestibule refers to a "dull red glow"; this is not something the player can TAKE, PUSH, OPEN or otherwise interact with as if it were a physical object, but it is plainly something the player can see, and so could EXAMINE. This makes it a good candidate for implementation as a Vaporous:

```
+ redGlow : Vaporous 'dull red glow' 'dull red glow'
    "It flickers a dull, hungry shade of red with a diabolical, fiery look to it. "
;
```

The only thing you can meaningfully do to a Vaporous is EXAMINE in (or LISTEN TO it or SMELL it if the author provides a listenDesc and a smellDesc), otherwise it provides one standard message if the player tries to LOOK IN, UNDER or BEHIND ("You just see the dull red glow.") and another if you try to do anything else to it, such as PUSH, TAKE or EAT it ("You can't do that to a dull red glow."). For most purposes these messages are probably fine, but you can easily change them if you want to by overriding the `lookInVaporousMsg` and `notWithIntangibleMsg` properties, e.g.:

```
+ redGlow: Vaporous 'dull red glow' 'dull red glow'
    "It flickers a dull, hungry shade of red with a diabolical, fiery look to it. "
    lookInVaporousMsg(obj) { return 'It\'s just as red whichever way you look at it. '; }
    notWithIntangibleMsg(obj) { return 'That\'s not a particularly practical suggestion. '; }
;
```

Note that these two message properties have to be overridden as methods with a single parameter that return a single-quoted string. Unfortunately, this is not a general rule when overriding a message property; in other cases you might simply need to override the message property with a single-quoted string. To find out what you need to do in any particular case you need to look up the corresponding message property in the library code to see how it is implemented.

13.6. SimpleOdor

Along with its close relative, [SimpleNoise](#), SimpleOdor offers a very straightforward way of representing an intangible sensory presence (i.e. a sound or smell), which can be associated either with a location, a particular object, or a number of objects. For more sophisticated behaviour involving smells you may want to consider the [Odor](#) class, which we shall come to shortly.

Let's suppose that we want the player to know that there's a whiff of sulphur around the cavern entrance we mentioned in the description of the hellVestibule location. We could achieve this simply by adding the following to the list of objects in hellVestibule

```
+ SimpleOdor 'sulphur/sulfur' 'sulphur'
    "A strong whiff of sulphur comes through the cavern entrance to the east. "
;
```

TADS 3 Tour Guide

Now if we go to hellVestibule and type a SMELL command, we'll see the message "A strong whiff of sulphur comes through the cavern entrance to the east. " This is the standard use of SimpleOdor (and SimpleNoise): to add an ongoing smell (or sound) to a location.

However, since we've mentioned the cavern entrance in the room description, and we say that that's where the smell is coming from, we might prefer to associate the SimpleOdor particularly with the entrance:

```
+ cavernEntrance: ThroughPassage 'east eastern cavern entrance' 'cavern entrance'
  "It's wide, with a sign next to it, and it emits an eerie red glow. "
;

++ SimpleOdor 'whiff sulphur/sulfur' 'sulphur'
  "A strong whiff of sulphur comes through the cavern entrance. "
;
```

At first sight this may seem to achieve no more than adding `smellDesc = "A strong whiff of sulphur comes through the cavern entrance. "` to the definition of `cavernEntrance`. There is, however, one small but important difference. Using the `smellDesc` method, we get a description of the odor if we type `SMELL CAVERN ENTRANCE` but not if we just type `SMELL` (in which case we'll be told "You smell nothing out of the ordinary"). Using the `SimpleOdor` object, however, we get the description of the smell either way. We can also `SMELL WHIFF` or `SMELL SULPHUR` or `SMELL WHIFF OF SULPHUR`.

We can also add a further refinement. With the `SimpleOdor` nested in the `cavernEntrance` as above, if we issue the command `SMELL RED GLOW` we'll be told "You smell nothing out of the ordinary. " Now, there's an argument for saying that since a glow doesn't smell, this is the right response. And yet one may feel it's a bit odd that when one smells the entrance one smells the sulphur, but that when one smells the glow coming through the entrance one smells nothing odd; is the human sense of smell really that localised? Well, if we did take that view we could very easily attach the `SimpleOdor` to the red glow as well by making it a `MultiLoc`:

```
+ MultiLoc, SimpleOdor 'whiff sulphur/sulfur' 'sulphur'
  "A strong whiff of sulphur comes through the cavern entrance. "
  locationList = [cavernEntrance, redGlow]
;
```

(Note that once we define it as a `MultiLoc` the `+` no longer defines its location, it just allows us to continue listing other objects after it with the `+` notation and have them be located in `hellVestibule`). Now we can `SMELL`, `SMELL ENTRANCE`, `SMELL WHIFF`, or `SMELL RED GLOW` and receive the same answer each time, and this is certainly rather more than we can do by setting one `smellDesc` property.

If we really want, we can take this a stage further still. `SimpleOdor` (and [SimpleNoise](#)) have an `isAmbient` property which, by default, is set to `true`. If we set it to `nil`, the `SimpleOdor` won't wait for us to `SMELL` anything, it'll announce its presence each time we get a description of the room - or each time we examine the cave entrance or the red glow:

```
+ MultiLoc, SimpleOdor 'whiff sulphur/sulfur' 'sulphur'
  "A strong whiff of sulphur comes through the cavern entrance. "
  isAmbient = nil
  locationList = [cavernEntrance, redGlow]
;
```

You may or may not think the effect is appropriate here; it calls attention to a smell that the player might otherwise miss, but could quickly become monotonous. We are getting to the point where it might be better to use the greater complexity of an [Odor](#) to get the effect we'd really like. Nevertheless, we introduce the `isAmbient` property here just to show what it does; whether you prefer to leave `isAmbient` as `nil` or `true` here is up to you. It also illustrates a point that may at first sight seem counter-intuitive. A `SimpleOdor` (or any other kind of `SensoryEmanation` object) announces its presence when `isAmbient` is `nil`, but not when it is `true`, whereas one might have expected an ambient sensory one to be something that was *more* proactive in making its presence felt. Here we have to understand 'ambient' in the sense of 'background' (which is admittedly not quite the meaning given in the OED); a background smell (or noise) is one that doesn't obtrude itself on our awareness unless and until we actively seek it out (by explicitly `SMELL`ing or `LISTEN`ing).

13.7. SimpleNoise

SimpleNoise is simply the sonic equivalent of [SimpleOdor](#). Everything that applies to the one applies to the other, except for the obvious difference that a SimpleNoise response to LISTEN or LISTEN TO SOMETHING commands rather than SMELL and SMELL SOMETHING commands.

Having taken SimpleOdor through its paces and exercised it every which way, one simple example of a SimpleNoise should suffice. Add the following directly after the definition of the caveEntrance object:

```
++ SimpleNoise 'intermittent muffled deep rumbling sound/sounds' 'sound'
    "An intermittent and slightly muffled deep rumbling echoes through the cavern entrance. "
;
```

In this case there seems to be no need to attach this sound to the red glow as well (one might conceivably smell the glow, but one would not expect to hear it), and besides there is no need to go over essentially the same theme and variations in a closely related key. For more sophisticated sounds, though, you might want to consider using the Noise class.

13.8. Odor

The Odor class is the big brother of the [SimpleOdor](#) class we met just a little earlier. It's function is to represent a smell emanating an object or pervading a location, but in a way that gives the author more control than does the SimpleOdor. To make an object the source of an odour (or noise), simply locate the odour (or noise) within that object.

An Odor (or a [Noise](#) for that matter) provides several properties for customisation:

sourceDesc - The description of the odour (or noise) that's added to the description of the source of the odour (or noise) when the source is examined.

descWithSource - The description of the odour/noise given when the source is visible and we EXAMINE or SMELL the odour (or listen to the noise).

descWithoutSource - The description of the odour/noise when the source is not visible (e.g. because it's in a closed container or hidden under something else)

hereWithSource - The message displayed in the room description to describe this smell or sound when its source is visible.

hereWithoutSource - The message displayed in the room description to describe this smell or sound when its source is not visible.

noLongerHere - The message displayed when the source of the sound or smell goes out of scope (e.g. because the player character leaves the location of its source)

isAmbient - As with SimpleNoise, this is a true/nil property that decides whether the smell mentions itself in room descriptions etc. (if isAmbient is nil) or only in response to an explicit SMELL/EXAMINE command (if isAmbient is true). The only difference from simpleOdor (or SimpleNoise) is that on Odor (and Noise) isAmbient is nil by default.

displaySchedule - This can contain a list of numbers defining the frequency with which the smell (or noise) is mentioned spontaneously. This can be used to emulate the fact that once someone becomes used to a sound or smell it tends to fade into the background of their awareness, and to avoid the repetitiveness that might come from displaying the same message about the smell (or noise) each turn. For example, if displaySchedule were set to [1, 2, 4] a message describing the smell would be displayed for two successive turns, then again after two turns, then every four turns thereafter. If the list of numbers end with nil the spontaneous display of messages about the smell ceases when the end of the list is reached.

isEmanating - This is a boolean flag (true or nil) that can be used to turn the odor (or sound) on or off. The default value is true (i.e. the SensoryEmanation is on). For SensoryEmanations belonging to dynamic objects such as actors, it can be useful to turn the emanation on or off; e.g. an actor might stop humming or hammering when the player character converses with him, but resume making humming or hammering noises once the conversation is over.

TADS 3 Tour Guide

So much for the theory, now let's put it into practice. First we'll create another location:

```
hellFireCavern : Room 'Hell Fire Cavern' 'Hell Fire Cavern'
    "This narrow shelf of rock ends at a round hole to the west, and a sheer
    drop to the east. It overlooks a bleak and barren plain several hundred
    feet below, on the far side of which an ugly volcano, the aptly-named Mount Gloom,
    belches fire and smoke and ash in constant rotation, shedding a hellish red
    light over the entire infernal scene. A rough staircase of sorts, in places
    little more than a slippery stone pathway and in others a jumble
    of rocks, leads northwards down to the lava-strewn plain. "
    west = cavernExit
    out asExit(west)
    north = roughStaircase
    down asExit(north)
    south : NoTravelMessage { "That way is solid rock. " }
    east : NoTravelMessage { "There's a sheer drop of several hundred feet that way. " }
;

+ cavernExit : ThroughPassage ->cavernEntrance 'round hole' 'round hole'
    "The large round hole piercing the cavern wall is easily large enough
    to walk through."
;

+ roughStaircase : StairwayDown 'rough stone slippery staircase/pathway' 'rough staircase'
    "It looks a rough descent, possibly treacherous in places, but probably
    passable with care. "
    canTravelerPass(traveler) { return traveler.isMasked; }
    explainTravelBarrier(traveler)
        { "The sulphurous fumes become too overpowering and drive you back. "; }
;

MultiLoc, Distant 'mount volcano/gloom' 'volcano'
    "The volcano rises up from the basalt plain like an angry sore, belching fumes,
    smoke and occasional balls of lava, which spit from the summit and ooze
    pus-like down its rugged slopes. "
    locationList = [hellFireCavern]
;

+ Vaporous 'hellish red light' 'hellish red light'
    "It's the sole source of light for the great cavern, enough to see by, but
    only in a gloomy, bloodshot sort of way. "
    smellDesc = "You can't exactly smell the light, but you can sure
    smell the sulphur! "
;
```

There's nothing new here, but note the use of another Vaporous object to represent the hellish red light. We make the volcano a MultiLoc since it will be visible from more than one location, but we can't add the other locations to its locationList until we've created them. For now, we'll concentrate on making in the source of the sulphurous smells by adding:

```
+ Odor 'strong smell sulphur/sulfur' 'smell of sulphur'
    descWithSource = "The sulphurous fumes are almost certainly coming from
    the volcano. "
    hereWithSource = "There's a strong smell of sulphur, almost enough to choke you. "
    displaySchedule = [2, 3, 3, 5]
    noLongerHere = "The smell of sulphur diminishes a little. "
;
```

In order to descend the rough staircase the player character needs to be wearing a gas mask, which we'll provide in the next section. In anticipation of that, we have to consider what happens to all the odours when the player character is wearing the gas mask; presumably none of them should be reported. This is actually a little tricky to achieve.

We can start by preventing any actor from smelling an object if that actor is wearing the gas mask:

```
modify Actor
    canSmell(obj)
    {
        if(isMasked)
            return nil;
        else
```

TADS 3 Tour Guide

```
        return inherited(obj);
    }
;

```

That deals with the transitive form of the smell command, such as SMELL SULPHUR, but we also need to deal with the intransitive form, the simple SMELL command:

```
modify SmellImplicitAction
    execAction()
    {
        if (gActor.isMasked)
        {
            "{You/he} can't smell anything with that gas mask on. ";
        }
        inherited;
    }
;

```

Finally, if the player character is wearing the gas mask we also have to block the messages that will be displayed in the room description or according to the display schedule, and the message that's displayed when an Odor goes out of scope:

```
modify Odor
    emanationHereDesc
    {
        if (gPlayerChar.isMasked)
            return;
        inherited;
    }
endEmanation
{
    if (gPlayerChar.isMasked)
        return;
    inherited;
}
;

```

Note that so far we have not referred to any specific gas mask object; we have simply referred throughout to as yet undefined Actor property `isMasked`. This not only lets us compile the code before defining any gas mask object (`isMasked` will simply return `nil`), it means that when we come to define what counts as wearing a gas mask, we need only do so in one place (`Actor.isMasked`); moreover, if we subsequently want to change what counts as wearing a gas mask, we need only change it in one place.

13.9. Noise

The Noise class works precisely like the [Odor](#) class, except that it is used for sounds rather than smells. Precisely the same properties are available to customise it. We can continue our illustration by making the volcano the source of a sound as well as a smell:

```
+ Noise 'ominous sound/rumble/rumbling' 'rumble'
    sourceDesc = "Mount Gloom seems to be the source of the ominous rumbling. "
    descWithSource = "The continuous bass rumble is punctuated by percussive
        explosions at irregular intervals. "
    hereWithSource = "An ominous rumble shakes the vast cavern. "
    displaySchedule = [1,2,2,4]
;

```

The only thing we have done new here is to add a `sourceDesc`, which you should see added to the description of the volcano when you EXAMINE MOUNT GLOOM.

To illustrate the `descWithoutSource` and `hereWithoutSource` properties we'll go back and add a ticking sound to the [bomb](#) we buried under a pile of rubble some time back, at the same time expanding what the bomb does when it explodes.

TADS 3 Tour Guide

```
+ bomb : Hidden, Immovable 'unexploded bomb/cylinder' 'bomb'
  "It's a fat, round-nosed cylinder with tail fins, on a couple of which
    are painted tiny swastikas. "
  discover()
  {
    inherited;
    new SenseFuse(self, &explode, 3, self, sight);
  }
  explode()
  {
    "The bomb explodes, the blast sending chunks of masonry flying in all
      directions, one piece of strikes you square on the head. ";
    if(gPlayerChar.isIn(location))
      endGame(ftDeath);
    respiratorBox.moveInto(location);
    respiratorBox.moved = nil;
    moveInto(nil);
  }
  cannotTakeMsg = 'You must be joking! '
  cannotPushMsg = 'That might set it off. '
  cannotMoveMsg = 'It\'s probably safest to leave it just where it is. '
;

++ Noise 'tick/ticking' 'ticking'
  sourceDesc = "It's ticking. "
  descWithSource = "The ticking is coming from the bomb. "
  descWithoutSource = "The ticking seems to be coming from the pile of rubble. "
  hereWithSource = "The bomb is ticking. "
  hereWithoutSource = "A ticking comes from the direction of the rubble. "
  displaySchedule = [1]
;

respiratorBox : OpenableContainer 'small (respirator) khaki bag/box' 'khaki bag'
  "The square bag is made of coarse khaki fabric and has a pair of carrying straps. "
  bulkCapacity = 4
  initSpecialDesc = "A small khaki bag lies in the street, perhaps dislodged from the
    rubble by the recent explosion. "
;

+ gasMask : Wearable 'gas mask/respirator/gas-mask/gasmask' 'gas mask'
  "It's an ungainly-looking thing with round glass circles for seeing through
    and a kind of cylindrical snout to fit over nose and mouth, all held together
    by a black rubber face-mask. "
;
```

The descWithoutSource and hereWithoutSource properties contain what is displayed while the bomb is still hidden in the rubble. Once the player investigates the source of the tick by looking in or under the rubble, the bomb is revealed and the descWithSource and hereWithSource messages are used instead. We set the displaySchedule to [1] to display the hereWith/WithoutSource message each turn, since the ticking can reasonably be expected to engage the player's attention.

Now that we've defined the gas mask, we can (provisionally) define what it means for an actor to be masked:

```
modify Actor
  canSmell(obj)
  {
    if(isMasked)
      return nil;
    else
      return inherited(obj);
  }
  isMasked = (gasMask.isWornBy(self))
;
```

13.10. SenseConnector

Unless the game author takes steps to make things otherwise, each location behaves like a sealed island; nothing that happens in one location can be seen, heard, felt, or smelled in another. Usually this is realistic enough, but there are occasions when it's not quite what we want; on such occasions we can join two or more locations by a `SenseConnector`, which can pass one or more senses between locations with varying degrees of transparency. The [DistanceConnector](#) we have already met is a specialized kind of `SenseConnector` that passes all four senses (taste excluded as not really relevant) as *distant*.

There are basically two ways you can define what senses a `SenseConnector` passes in what way. The first way is to set its **connectorMaterial** property to one of the materials defined in the library (or one you define yourself), each of which defines some combination of the senses as *transparent* (which means that they are passed with no degradation, as if their source was right under our nose) and the remainder as *opaque* (which means that they aren't passed at all). Alternatively, if none of these pre-defined materials give you what you want, and you don't want to define another (which may be needless labour), you can override your `SenseConnector`'s **transSensingThru(sense)** method to return the appropriate level of transparency for each sense, which may be either *transparent*, *opaque*, *distant* or *obscured*. This is the method we shall be using shortly.

We have arranged for the bomb to detonate and to kill the player character if s/he's nearby, but as yet there's nothing to tell the player when the bomb explodes if the player character goes wandering off into a neighbouring location. There are several ways this could be achieved, including the brute force method of (say) a [prompt daemon](#) that checks once a turn whether the bomb is still present and reports the explosion if it is not, giving a different message according to the location of the actor, and then removing itself from the list of active events, something like this:

```
+ bomb : Hidden, Immovable 'unexploded bomb/cylinder' 'bomb'
  "It's a fat, round-nosed cylinder with tail fins, on a couple of which
  are painted tiny swastikas. "
  discover()
  {
    inherited;
    new SenseFuse(self, &explode, 3, self, sight);
    daemonID = new PromptDaemon(self, &daemon);
  }
  daemon()
  {
    if (bomb.isIn(nil))
    {
      switch (gPlayerChar.location)
      {
        case streetJunction: "Loud Bang!"; break;
        case road : "Distant Bang!"; break;
        case shop : "Muffled Bang!"; break;
      }
      daemonID.removeEvent;
      daemonID = nil;
    }

    }
  daemonID = nil
  ...
;
```

This approach works, and could also have been used to handle the case where the player character is in the same location as the bomb when it goes off. We shall, however, explore a different approach that models the situation in a less ad-hoc way, and which perhaps is more easily extensible. We'll be making the explosion of the bomb create a `SoundEvent` that can be heard someway off; but to allow the `SoundEvent` to be sensed at other locations we need to connect them by a `SenseConnector`. For this purpose we'll assume that sound is the only thing that will be transmitted (we can't actually see, smell or feel what's going on near the bomb unless we're there), and that as the other locations are a little way away, any sound should be passed as *distant* rather than *transparent*. We can then define our `SenseConnector` thus:

```
SenseConnector, Intangible
  transSensingThru(sense)
  {
    if (sense==sound)
      return distant;
    else
```

TADS 3 Tour Guide

```
        return opaque;
    }
    locationList = [londonStreet, streetJunction, road, shop]
;
;
```

Note that we need to give the SenseConnector some other class as well. Here we make it an Intangible, since it doesn't represent any tangible object in the game, but in other situations you might want your SenseConnector to be a physical object like a door, wall, or window that actually connects two locations. We use the locationList property to list the locations we want connected (not all of which we have defined yet).

If you try to run this as things are, apart from the undefined location, you'll encounter another problem: when the player character moves north from londonStreet to streetJunction s/he can now still hear the ticking from the bomb. This probably isn't very realistic - one might expect to hear the bomb exploding from some way away, but not ticking. Fortunately, this is very easy to fix: because we have defined the SenseConnector as passing sounds through a distance, it won't pass any whose **soundSize** is set to *small* (as opposed to *medium* - the default, or *large*). We thus just need to add this refinement to the definition of our Noise object:

```
++ Noise 'tick/ticking' 'ticking'
    sourceDesc = "It's ticking. "
    descWithSource = "The ticking is coming from the bomb. "
    descWithoutSource = "The ticking seems to be coming from the pile of rubble. "
    hereWithSource = "The bomb is ticking. "
    hereWithoutSource = "A ticking comes from the direction of the rubble. "
    displaySchedule = [1]
    soundSize = small
;
;
```

13.11. SensoryEvent

So far we have added a means of passing sound from the bomb to neighbouring occasions, but we've yet to create a sound to pass. Unlike the ticking of the bomb, which is continuous (until the bomb detonates), its explosion is one-off (and dramatic). We can represent it with a SensoryEvent, or more specifically, a SoundEvent (one of the subclasses of SensoryEvent along with SightEvent and SmellEvent). The definition of the SoundEvent could hardly be simpler:

```
explosionEvent : SoundEvent;
```

There *are* a couple of properties you can play with on this class if you really want to: the **sense** property contains the sense in which the event is observable, but it's fairly obvious that for a SoundEvent this can only be sound (as the library indeed defines it); to define an event as a SoundEvent and change its sense property to smell would be perverse, confusing, and, well, pretty senseless. You might have occasion to define something different here if you wanted an event making use of a different kind of sense you had defined yourself, such as a burst of microwaves to be picked up by a radar receiver. The other property of interest is **notifyProp**, which contains a pointer to the property to be notified (i.e. the method to be called) on all objects in range of the event. For a SoundEvent this is defined as ¬ifySoundEvent. Again there is no real need to change it; if you had a large number of SoundEvents which might affect the same group of objects you *might* want to give them all a different notifyProp so that they'd call different methods, but there's no need to do this, since (as we shall see shortly) the notifySoundEvent method (or the corresponding methods for SmellEvents and SightEvents) can tell from their parameters what event has triggered them, so it's probably best to leave this property alone unless you're creating handling for a new kind of SensoryEvent (such as a custom RadiationEvent).

In order to make a SensoryEvent do anything, we simply need to call its **triggerEvent(source)** method, where *source* is the object that's notionally the source of the sound, light or smell that the event represents. In the case of our bomb, we simply need to put the appropriate call in the bomb's explode method:

```
+ bomb : Hidden, Immovable 'unexploded bomb/cylinder' 'bomb'
    "It's a fat, round-nosed cylinder with tail fins, on a couple of which
    are painted tiny swastikas. "
    discover()
    {
        inherited;
        new SenseFuse(self, &explode, 3, self, sight);
    }
;
```

TADS 3 Tour Guide

```
explode()
{
    "The bomb explodes, the blast sending chunks of masonry flying in all
    directions, one piece of strikes you square on the head. ";
    if(gPlayerChar.isIn(location))
        endGame(ftDeath);
    respiratorBox.moveInto(location);
    respiratorBox.moved = nil;
    explosionEvent.triggerEvent(self);
    moveInto(nil);
}
cannotTakeMsg = 'You must be joking! '
cannotPushMsg = 'That might set it off. '
cannotMoveMsg = 'It\'s probably safest to leave it just where it is. '
;
```

This is all very well, but in order for this to have any effect, we need to have something that responds to the event when it's triggered. Such a something is called a *SoundObserver* (or *SightObserver* for a *SightEvent*, or *SmellObserver* for a *SmellEvent*). This is defined as a mix-in class, so we could, for example, mix it in with a *SecretFixture* in each of the locations where we want the explosion to be reported (i.e. all those joined to *londonStreet* by the *SenseConnector*). When the *triggerEvent* method is called, it should in turn call the **notifySoundEvent(event, source, info)** method of every *SoundObserver* object within hearing range. The *event* parameter is the *SoundEvent* that's just been triggered, while the *source* is the object that's the notional source of the sound (i.e. the source specified as the paramant in the *explosionEvent.triggerEvent(source)* call).

As a first (but erroneous) attempt, we might try something like this in *streetJunction*:

```
+ SoundObserver : SecretFixture
    notifySoundEvent (event, source, info)
    {
        if(source == bomb)
            "\nThere is a loud explosion and a cloud of dust from the street to
            the south as the bomb explodes amongst the rubble. "
    }
;
```

The test for *source* being *bomb* is not strictly necessary here, since there's only one *SoundEvent* that's going to be fired round here, but it's a good idea to include it, not only to show how one might apply such a test, but also in case at some later stage you wanted to add more *SoundEvents*, and needed to be sure that the right one was going to be responded to on any occasion.

But there are two things wrong with the definition we have given above (though you'll only encounter one of them if you try it). The first is that the message will never be displayed (this is the problem you'll encounter), and the second is that, if it were displayed, this wouldn't necessarily be what we wanted, since what we actually want is to see a message appropriate to the location in which the player character is located at the time, not every message from every *SoundObserver* that defines a response to this *SoundEvent* no matter where the player character is located. Fortunately, we can quite readily fix both problems at once.

The reason no message is displayed may seem quite mysterious, and very hard to track down it, for example, you try to trace what's happening through the debugger. It is, in fact, that *explosionEvent.triggerEvent* is called by a *SenseFuse*, and the *SenseFuse* makes sure that no messages are displayed unless the object associated with the fuse is in scope for the player character. The whole chain of events from exploding the bomb to triggering *explosionEvent.triggerEvent* to calling *notifySoundEvent* takes place in the context of the *SenseFuse*, and so the message displayed in *notifySoundEvent* will not be displayed. The solution is to have *notifySoundEvent* use *callWithSenseContext* to set up the visual sensory context of the location where we've placed the *SoundEvent*. This also ensures that the player sees only the message revelant to the location of the player character. Since we shall be deploying several of these receptors it's worth putting this extra bit of complication into a class which we can reuse as needed:

```
class BangObserver : SecretFixture
    notifySoundEvent (event, source, info)
    {
        if(source==bomb)
            callWithSenseContext(self, sight, {: bang});
    }
    bang = "Bang! "
;
```

TADS 3 Tour Guide

Note that BangObserver does *not* inherit from SoundObserver. There's no reason why it shouldn't, but also no reason why it needs to, since all SoundObserver does is define an empty notifySoundEvent method (which we're redefining here anyway).

Now all we need to do is to add a BangObserver in streetJunction:

```
streetJunction : OutdoorRoom 'Street Junction' 'the junction'
    "The street from the south meets another running east-west. A short way down
    to the street to the east a fire crew is fighting a blazing fire. "
    south = londonStreet
    east : FakeConnector { "After taking a few steps east you recall that
        discretion is the better part of valour and decide to keep out of the
        way of the fire crew. "}
    west = road
    ...
;

+ BangObserver
    bang = "\nThere is a loud explosion and a cloud of dust from the street to
    the south as the bomb explodes amongst the rubble. "
;
```

At the same time we add a new connexion to the west, so we need to define the appropriate location:

```
road : OutdoorRoom 'Road' 'the road'
    "The road runs west from the junction passed a row of terraced houses and shops.
    It seems strangely deserted, perhaps because the air-raid has sent everyone
    scuttling into shelters. A shop suffering bomb-damage stands open to the north. "
    east = streetJunction
    west : FakeConnector { "After a few steps down the street you decide that
        wandering too far round the city in the middle of an air-raid might be
        bad for your health. "}
    north = shop
    in asExit(north)
;

+ BangObserver
    bang = "\nThere is a muffled explosion from the southeast. "
;

+ Enterable ->shop 'shop' 'shop'
    "The damage to the shop, probably a grocer, looks substantial; not only have the
    windows been blown in but much of the surrounding brickwork with it. From the
    outside, at least, the shop looks abandoned. "
;
```

And then, in turn, the interior of the shop:

```
shop : Room 'Grocery Shop' 'the grocery shop'
    "The interior of the shop confirms the impression conveyed by the exterior: the
    bomb damage here is substantial; broken glass, shattered masonry and dust
    cover the floor, and the stock has all been removed, leaving nothing but a
    bare counter. "
    out = road
    south asExit(out)
;

+ BangObserver
    bang = "There's a sudden explosion from somewhere nearby outside, bringing
    more dust cascading from the ceiling. "
;

+ Decoration 'broken shattered glass/masonry/dust/debris' 'debris'
    "Shards of glass mingle with bits of brickwork all over the floor, and a
    patina of coarse white dust covers everything. "
;
```

The usefulness of the SoundEvent/SoundObserver implementation now becomes apparent, since one can keep adding suitable BangObserver objects within each location that might be affected.

TADS 3 Tour Guide

The purpose of this trip to London in the Blitz is to collect not one but two gas masks; two will be needed because our intrepid player character will eventually be given a travelling companion (Sarah) who'll also need a gas mask if she is to venture into Hell Fire Cavern with him (let's assume it's a him from now on, for ease of reference). The two gas masks will be functionally identical, and there's no reason to suppose that two gas masks picked up in wartime London will look very different from each other, so rather than impose any artificial distinctions, we'll create a GasMask class and make two interchangeable instances of it:

```
class GasMask : Wearable, Hidden 'gas mask/respirator/gas-mask/gasmask
    *masks*gasmasks*respirators'
    'gas mask'
    "It's an ungainly-looking thing with round glass circles for seeing through
    and a kind of cylindrical snout to fit over nose and mouth, all held together
    by a black rubber face-mask. "
    isEquivalent = true
    dobjFor(Wear)
    {
        verify()
        {
            inherited;
            if(gActor.isMasked)
                illogicalNow('{You/he} {is} already wearing a gas mask. ');
        }
    }
;
```

We set the **isEquivalent** property to true to indicate that all members of the class are interchangeable (like the [candles](#) and [matchsticks](#) we implemented earlier), and override the verifyDobjWear method to allow only one gas mask to be worn by any one Actor at a time. The definition of the gas mask found in the respiratorBag then becomes simply:

```
respiratorBox : OpenableContainer 'small (respirator) khaki bag/box' 'khaki bag'
    "The square bag is made of coarse khaki fabric and has a pair of carrying straps. "
    bulkCapacity = 4
    initSpecialDesc = "A small khaki bag lies in the street, perhaps dislodged from the
    rubble by the recent explosion. "
;

+ gasMask1 : GasMask
    discovered = true
;
```

Note that we have renamed it gasMask1 to distinguish it from the second gas mask we'll now add (continuing the definition of the shop):

```
+ RearContainer, Fixture 'long wooden counter' 'counter'
    "Like everything else round here, the long wooden counter is bare apart from a
    thick covering of dust. "
;

++ gasMask2 : GasMask
    initSpecialDesc = "A gas mask lies abandoned on the floor behind the counter. "
;
```

It may seem strange to derive GasMask from the Hidden class as well as from the Wearable class when we actually want only one of the gas masks to be hidden. But this is the only way we can make both gas masks exactly equivalent, and it only involves us in adding a single line of code (revealed = true) to the definition of gasMask1.

This now leaves us with another task: changing the definition of Actor.isMasked (aren't you glad now we defined it so we'd only have to change it one place) to allow any gas mask to do the job:

```
modify Actor
    canSmell(obj)
    {
        if(isMasked)
            return nil;
        else
            return inherited(obj);
    }

/* if the first parameter is nil or not supplied, return the first gas mask
```


TADS 3 Tour Guide

```
* among the actor's possessions. If the first parameter is true,
* return the gas mask (if any) that is being worn by the actor.
*/
maskObj([params])
{
    local worn = nil;
    if(params.length>0)
        worn = params[1];
    foreach(local cur in contents)
    {
        if(cur.ofKind(GasMask) && (worn==nil || cur.isWornBy(self)))
            return cur;
    }
    return nil;
}
isMasked
{
    return maskObj(true) != nil;
}
;
```

We could have used a slightly simpler definition such as `isMasked = (gasMask1.isWornBy(self) || gasMask2.isWornBy(self))`, but this approach would soon become tedious if we decided to add any more gas masks, so we have used a more general approach that checks whether there's any GasMask object among the actor's possessions that's currently being worn by the actor. This would allow us to add more gas masks into the game if we wanted them without having to worry about doctoring the definition any further. We have defined `maskObj()` as a separate method rather than incorporating it directly into `isMasked()`, since we'll be needed `maskObj()` later.

14. Attachables

14.1. Attachables - Overview

On occasion the need will arise to attach one object to another. Quite often this can lead to complex situations. If I attach the rope to the chair and walk off carrying the rope, what should happen? Does the rope break, the chair topple over, or the rope constrain me from walking any further? If I attach the red lego brick to the yellow one and pick up the red brick, does it become detached from the yellow one or bring it along with it? If I plug the iron into the wall and attempt to walk away from it, does the plug pull out of the socket, the iron out of my hand, or am I brought up short?

Because the possible permutations are so complex the TADS 3 library can hardly cover every eventuality, but rather than leave the game author with nothing but a few basic verb definitions, it does provide a few classes that at least provide a framework for attaching and detaching objects:

```
Attachable
  NearbyAttachable
  PermanentAttachable
  PermanentAttachmentChild

PlugAttachable
```

14.2. Attachable

The Attachable class makes the handling of attaching and detaching objects to and from each other a bit easier, but it does also leave quite a bit to the game author to implement. Where the Attachable class helps is first in making the ATTACH command symmetrical (so that ATTACH A to B is handled the same as ATTACH B to A) and secondly in providing a framework for specifying precisely what happens when two objects are ATTACHED or DETACHED. To work with Attachables successfully, it is important to understand both this framework, and also what handling the Attachable class does and does not itself provide.

Firstly, an Attachable object defines an **attachedObjects** property which contains a list of the other objects to which it is attached. This list is automatically maintained on both objects involved in an ATTACH X TO Y or DETACH X FROM Y command. Moreover the method **isAttachedTo(obj)** can be used to test whether an Attachable object is currently attached to obj.

An Attachable object also defines a method **canAttachTo(obj)** which determines whether the Attachable can be attached to obj. For this to return true, obj must also be an Attachable, but one that overrides canAttachTo(obj) to allow it to be attached to the first Attachable. To make this doubtless confusing explanation a bit clearer, this means that if you want to be able to ATTACH X TO Y then both X and Y must be of class Attachable, and *either* X or Y must override canAttachTo(obj) to return true when obj is the other object. So you must either override X.canAttachObj(obj) to return true when obj==Y, or override Y.canAttach(obj) to return true when obj==X. If X is the only object that can be attached to Y, then you could do this with:

```
X : Attachable, Thing ...
    ....
    canAttachTo(obj) { return obj == Y; }
    ...
;
```

Note that Attachable is a mix-in class, so we must also add Thing or some Thing-derived class to the class list.

The **canDetachFrom(obj)** method is similar, except that it generally allows X to be detached from Y unless either X.canDetachFrom or Y.canDetachFrom has been overridden to prevent it, or either X or Y is of class PermanentAttachment (or **isPermanentlyAttachedTo(obj)** has been overridden with some other condition).

In the event of either attachment or detachment being disallowed, the method **explainCannotAttachTo(obj)** or **cannotDetachMsg(obj)** is called; these can be overridden with customised messages if desired.

TADS 3 Tour Guide

It is important to realize (a) that this is about as far as the library takes it and (b) that this is never enough. As things stands, if you bring X into a location and attach it to Y which you find there, you can walk all over the map carrying X while X is considered attached to Y and Y remains where it is (and out of scope until to return to its location). It is hard to conceive of a situation where this would be what you actually want.

Because the library can hardly guess what you do want (should Y be dragged along with X, or detach itself from X, or what?) it is left up to you to define the rest of the behaviour, but the library does define four (by default empty) methods to help you do this: **handleAttach(other)**, **moveWhileAttached(movedObj, newCont)**, **handleDetach(other)** and **travelWhileAttached(movedObj, traveler, connector)**. The first of these is called on both objects in an ATTACH command (so need normally be defined on one of them) and the last on both objects in a DETACH command (so again need only be defined on one of them); **moveWhileAttached(movedObj, newCont)** is called on movedObj and every object attached to it when movedObj is about to be moved into newCont. The final case arises, for example, when walking into a room, attaching X to Y, and then walking away with X, since while X remains held by the player character, it does not change container; although in the plain English sense X is being moved when you walk away with it, it is not being moved in the sense that would result in a call to **moveWhileAttached**. To deal with this situation we use **travelWhileAttached**.

Not all attachment relationships are symmetrical. If we attach a red wire to a green wire it doesn't much matter if the red wire is described as attached to the green wire or the green to the red, but if we attach a flag to a battleship, we expect to see the flag described as attached to the battleship, not the battleship to the flag. To handle this Attachable provides an **isMajorItemFor(obj)** method, which should return true on the major item when obj is the minor item; for example you might define

```
battleship : Attachable, Enterable
...
isMajorItemFor(obj) {return obj==flag; }
;
```

But before we get too deep into such complications, we'll start with a very simple case. You'll recall that back on the north side of the lake we had a platinum ring and a diamond. We'll suppose that the diamond can simply be clipped back into the ring to form a diamond ring; in this case we simply want ATTACH DIAMOND TO RING or ATTACH RING TO DIAMOND to replace both the diamond and the ring with a single diamond ring object. We could put the handling on either object; we'll use the ring:

```
ring : Attachable, Thing 'platinum ring/band/recess' 'platinum ring'
"It's a plain platinum band, with a small empty recess on one side. "
location = dressingTable.subContainer
canAttachTo(obj) { return obj==diamond; }
handleAttach(other)
{
    diamondRing.moveInto(gActor);
    moveInto(nil);
    other.moveInto(nil);
    "{You/he} snap{s} the diamond back into its setting on the ring. ";
}
getFacets() { return [diamondRing]; }
;
```

Note that we start by adding Attachable to the start of ring's class list. When the diamond is attached to the ring we move the diamond ring into the actor performing the action, since it's reasonable to suppose that this is where it would end up unless the actor made a point of putting it down again. Finally, we override getFacets so that if we have just referred to the ring without the diamond, the pronoun 'it' will refer to the diamond ring after the transformation. We then need to make minimal changes to the diamond object:

```
diamond : Attachable, Thing 'sparkle' 'sparkle' @pathEnd
"It looks like a genuine diamond - and a valuable one too, exquisitely cut
and multifaceted. "
iobjFor(CutWith) { verify() { } }
initSpecialDesc = ""
...
renamed = nil
getFacets() { return [diamondRing]; }
;
```

Now we'll try something more complicated. Some way back we left our intrepid adventurer stranded on the south side

TADS 3 Tour Guide

of the chasm - or least, with no further to go once he's got there. We'll now suppose that the tunnel to the south leads to a steel door that can only be operated by pressing a button that's concealed behind a loose stone. The only problem is that the wires have been cut, so before the button will work it must be detached from its fitting and repairs made to the wires. We'll implement the button and its container as Attachables, and the wires as [NearbyAttachables](#).

The first step is to provide the passage south from the chasm and the location by the steel door:

```
chasmLedge : DarkRoom 'Ledge of Chasm' 'the ledge of the chasm'
  "A deep, wide chasm splits the ground immediately to the north of this
  narrow ledge, while a dark tunnel runs south. Another tunnel can be
  seen leading north from the far side of the chasm. "
  north = deepChasm
  inRoomName(pov)
  {
    return 'on the ' + (pov.isIn(deepChasm) ? 'south' : 'far') + ' ledge of the chasm';
  }
  south = tunnelFromChasm
;

+ tunnelFromChasm : ThroughPassage 'dark tunnel' 'dark tunnel'
  "The dark tunnel runs south from here. "
  noteTraversal(traveler) { "You walk a long way down the tunnel. "; }
;

tunnelEnd : DarkRoom 'End of Tunnel' 'the end of the tunnel'
  "The tunnel from the north comes to an abrupt end before
  <<blankSteelDoor.isOpen ? 'a large opening into a well-lit
  corridor to the south' : ' a blank steel door'>>. There is nothing
  else here but the rough stone walls. "
  north = tunnelToChasm
  south = blankSteelDoor
  brightness = (blankSteelDoor.isOpen ? 3 : 0)
  roomParts = [defaultFloor, defaultCeiling, tunnelEndWestWall, defaultEastWall]
;

+ tunnelToChasm : ThroughPassage -> tunnelFromChasm 'tunnel' 'tunnel'
  "The long narrow tunnel leads off to the north. "
;

+ blankSteelDoor : Door 'blank steel door' 'blank steel door'
  "The door <<isOpen ? 'has slid open out of sight' : 'is without handle,
  keyhole or any other visible mechanism'>>"
  openStatus { }
  checkDobjOpen() { "There's nothing on the door to get a purchase on. "; exit; }
  checkDobjClose() { "The door has slid out of sight. "; exit; }
  makeOpen(stat)
  {
    inherited(stat);
    "The steel door slides <<isOpen ? 'open' : 'shut'>>. ";
  }
;

tunnelEndWestWall : RestrictedContainer, RoomPart 'west rough stone wall*walls' 'west wall'
  desc = "<<freeStone.isIn(self) ? 'On closer inspection, one of the stones in
  the wall looks a bit loose': 'There\'s a small round hole in the wall where a stone
  has been removed' >>. "
  validContents = [freeStone]
;
```

We have made a special room part to hold the stone that needs to be moved to find the button; we make it a `RestrictedContainer` for that purpose. We override the `blankSteelDoor.openStatus` to display nothing, since 'It's open' or 'It's closed' is superfluous given the way we have described the door. We use the object name `freeStone` rather than `looseStone` since the latter has already been used as a method name:

```
+ freeStone : Thing 'loose stone' 'loose stone'
  "It's a rough, round stone, about the size of a grapefruit. "
  bulk = 2
  weight = 3
  moveInto(newDest)
  {
    if(location==tunnelEndWestWall)
```

TADS 3 Tour Guide

```
"Removing the stone reveals a small round hole behind. ";
inherited(newDest);
}
;
```

We override `freeStone.moveTo` so that if it's being moved out of the wall it announces the presence of the small round hole that's thereby revealed. The next problem we are going to have to sort out is how to make things visible and invisible according to what's where. The easiest way to deal with this is to make a series of nested containers, which we'll make open or closed according to whether the obscuring object is present or not. We make the hole in the wall a `RestrictedContainer` so that it can handle an attempt to `PUT BUTTON IN HOLE` or `PUT STONE IN HOLE`, but we place it inside another container that represents the presence or absence of the stone in the wall (so that hole is visible if and only if the stone is not in the wall):

```
+ Container, SecretFixture
  isOpen = (!freeStone.isIn(tunnelEndWestWall))
;

++ holeInWall : RestrictedContainer, Fixture
  'small round circular hole*holes' 'small round hole'
  isListedInContents = (sightPresence)
  bulkCapacity = 3
  validContents = [holeButton, freeStone]
  iobjFor(PutIn)
  {
    action()
    {
      if(gDobj==freeStone)
        replaceAction(PutIn, gDobj, tunnelEndWestWall);
      else if(gDobj==holeButton)
        replaceAction(AttachTo, gDobj, buttonFitting);
      else
        inherited;
    }
  }
;
```

The button is the most complex object we need to define here. It starts attached to a fitting that also acts as the container for the tiny hole containing the wires (which we'll come to in a minute), so we define `buttonFitting` as being in its list of attached objects from the start, and override `canAttachTo` to allow the `buttonFitting` as an attachment. The button can only be detached if it has first been unscrewed so we define an `unscrewed` property to keep track of this and then test for it in `canDetachFrom`. The `handleAttach` and `handleDetach` methods then define the specialized handling we need for attaching and detaching the button from its fitting, in the former case moving the button into the hole and setting `unscrewed` to nil, in the latter moving the button into the actor's inventory (on the assumption that it will come off into his hand), and in both cases displaying a suitable message. We also need to define the handling for the `UNSCREW` command (basically to set `unscrewed` to true and make a suitable report). While we're at it we make `SCREW` equivalent to `ATTACH BUTTON TO FITTING`, and `TAKE BUTTON` equivalent to `DETACH BUTTON FROM FITTING` if the button is attached to the fitting:

```
+++ holeButton : Attachable, Button 'small black button/casing' 'small black button'
  "It's in a circular casing<<isAttachedTo(buttonFitting) ? ' '
  : ', inside which is a screw thread'>>. "
  canAttachTo(obj) { return obj == buttonFitting; }
  handleAttach(other)
  {
    moveTo(holeInWall);
    unscrewed = nil;
    "{You/he} screw{s} the button into its fitting. ";
  }
  canDetachFrom(obj) { return obj == buttonFitting && unscrewed; }
  cannotDetachMsg(obj)
  {
    if(obj==buttonFitting)
      return 'It seems to be screwed in place. ';
    else
      return inherited(obj);
  }
  handleDetach(other)
  {
    moveTo(gActor);
  }
;
```

TADS 3 Tour Guide

```
"{You/he} remove{s} the button from its fitting, revealing a tiny hole behind. ";
}
attachedObjects = [buttonFitting]
unscrewed = nil
doObjFor(Unscrew)
{
    verify() { if(unscrewed) illogicalNow('It\'s already unscrewed. '); }
    action()
    {
        "{You/he} unscrew{s} the button. ";
        unscrewed = true;
    }
}
doObjFor(Screw) remapTo(AttachTo, self, buttonFitting)
doObjFor(Take) maybeRemapTo(isAttachedTo(buttonFitting),DetachFrom, self, buttonFitting)
doObjFor(Push)
{
    action()
    {
        if(isAttachedTo(buttonFitting) && blackWire.isAttachedTo(greenWire)
            && blackWire.isAttachedTo(redWire))
            blankSteelDoor.makeOpen(!blankSteelDoor.isOpen);
        else
            "Nothing happens. ";
    }
}
;
```

At the end of all this we also define what happens when the button is pushed. Nothing happens at all unless the wires have been repaired (by attaching the black wire to both the red wire and the green wire to restore the connection) and the button is attached to its fitting; but if all these conditions are met we open or close the door (depending on whether it is currently closed or open).

The next step is to define the fitting to which the button is attached. Most of the attachment work has been done on the button, but we also need to make it a container that's opened when the button is removed so we can see what's inside:

```
+++ buttonFitting : Attachable, Container, Fixture '(button) fitting' 'fitting'
"<<isAttachedTo(holeButton) ? ' ' : 'It\'s circular, with a screw thread running
round the outside. '>>"
isOpen = (!holeButton.isAttachedTo(self))
isListedInContents = (isOpen)
;
```

We'll go on to implement the wires inside the fitting as [NearbyAttachables](#).

14.3. NearbyAttachable

A NearbyAttachable is an Attachable with some further specialization added. In particular, a NearbyAttachable enforces the condition that the attached objects must share the same immediate container. If one of the attached objects is moved out of their common container, the objects are automatically detached (this is enforced in NearbyAttachable.moveWhileAttached, which NearbyAttachable overrides). Conversely, if an ATTACH command is issued to a NearbyAttachable, one precondition of the action is that the direct object is in the same container as the indirect object; this means that when the command ATTACH X to Y is issued, if X is not already in the same container as Y, the attempt will be made to move it there as an implicit action. If the implicit action succeeds, the attachment is carried out, otherwise it fails.

This is precisely what we want with the wires behind the switch; for the black wire we need to attach to the red and green wires to be attached to them, it must be in the same container as those other two wires. Next we define that container, a tiny hole in the button fitting that's revealed when the button is detached:

```
++++ wireHole : RestrictedContainer, Fixture 'tiny square hole*holes' 'tiny hole'
"The tiny hole is just over an inch square. "
validContents = [redWire, greenWire, blackWire]
isListedInContents = true
;
```

TADS 3 Tour Guide

This definition should, of course, follow directly on from that of the `buttonFitting` object, so that the nesting works correctly. Next we define the red and green wires that go inside:

```
class ColouredWire : Attachable, Fixture
  desc = "\^<<theName>> protrudes from the <<LorR>> hand side of the tiny hole. It looks like
  it's been cut. "
  isListedInContents = true
  explainCannotAttachTo(obj)
  {
    if(obj.ofKind(ColouredWire))
      tooShortMsg;
    else
      inherited(obj);
  }
  tooShortMsg = "The red and green wires aren't quite long enough
  to reach each other. "
;

+++++ redWire : ColouredWire 'red wire*wires' 'red wire'
  LorR = 'left'
;

+++++ greenWire : ColouredWire 'green wire*wires' 'green wire'
  LorR = 'right'
;
```

Because the definition of the two wires is so similar we define a `ColouredWire` class for the common handling and then create a couple of instances of it. Finally, we need the piece of black wire that will join up the other two:

```
blackWire : NearbyAttachable, Thing 'short piece black wire*wires' 'short piece of black wire'
  @smallHoleInWall
  "It's about an inch long, covered with black insulation, but with the naked
  wire showing at each end. "
  canAttachTo(obj) { return obj is in (redWire, greenWire); }
;
```

Because `NearbyAttachable` (unlike `Attachable`) already enforces most of the special conditions we need, the only thing we need to define on the `blackWire` is the list of objects that it can be attached to (alternatively, we could have added `canAttachTo(obj) { return obj == blackWire; }` to the definition of the `ColouredWire` class). We let the black wire start life in the small hole in the wall behind the mirror in the cave, so it's not immediately obvious what it's for. We'll also be sneaky and make it of class `Hidden`, since it's quite feasible that the player wouldn't notice it unless he deliberately looks in the hole (and it also stops the wire announcing its presence before we're ready for it!).

That's all we need to do to get the button to work, but at the moment there's nothing on the other side of the door, which won't make it that rewarding to open. We can rectify that forthwith:

```
brightCorridor : Room 'Brightly-lit Corridor'
  "The corridor continues both east and west from here<<openDoorway.isOpen ?
  ', and there is an open doorway to the north' : ''>>. A large sign on the
  wall points west towards the Museum Entrance and east towards the Exit. "
  north = openDoorway
;

+ Decoration 'large sign' 'large sign'
  "MUSEUM ENTRANCE --&gt;\n&lt;-- WAY OUT\n"
;

+ openDoorway : SecretDoor -> blankSteelDoor 'open doorway' 'open doorway'
  "<<isOpen ? 'The door has slid open, revealing a dark tunnel beyond'
  : 'It\'s a featureless sheet of grey steel'>>. "
;
```

The `openDoorway` could almost have been made a `ThroughPassage`, since there's no way to approach it except from `tunnelEnd` when its other side is open, but we'll make it a `SecretDoor` just in case; there's no way of opening or closing it from this side, but at least it'll be open and closed in sync with its other side.

14.4. PlugAttachable

PlugAttachable is a mix-in class for use with Attachable or NearbyAttachable to make PLUG X IN Y behave like ATTACH X TO Y and UNPLUG X FROM Y behave like DETACH X FROM Y.

To demonstrate this, we'll impement a vending maching just down the corridor that needs to be plugged into the wall with an electrical cable before it'll dispense entrance tickets. First we'd better create the location for the machine:

```
brightCorridor : Room 'Brightly-lit Corridor'
  "The corridor continues both east and west from here<<openDoorway.isOpen ?
  ', and there is an open doorway to the north' : ''>>. A large sign on the
  wall points west towards the Museum Entrance and east towards the Exit. "
  north = openDoorway
  east = museumExit
;

museumExit : Room 'Museum Exit' 'the museum exit'
  "The brightly lit corridor from the west comes to an end here, by a door
  that looks firmly closed. Some sort of vending machine stands next to
  one wall, and there's an electrical outlet near to it at floor level. "
  west = brightCorridor
  east = museumExitDoor
;

+ museumExitDoor : IndirectLockable, Door 'door' 'door'
  "The door is marked EXIT. "
;
```

In this case we've made the museumExitDoor an IndirectLockable because we intend it never to be unlocked; it forms a boundary to the game world.

Now we need to add the electrical socket and the vending machine, both of which will need to be both PlugAttachable and Attachable as well their appropriate Thing-derived class:

```
+ socket : PlugAttachable, Attachable, Fixture
  'electrical outlet/socket' 'socket'
;

+ vendingMachine : PlugAttachable, Attachable, Heavy
  'vending machine' 'vending machine'
  "It's a machine for selling museum tickets. Near the top is a small vertical
  slot labelled <<vendingSlot.label>>, and lower down is the hole that
  presumably dispenses the tickets. "
  powerOn = (powerCable.isAttachedTo(self) && powerCable.isAttachedTo(socket))
  afterAction()
  {
    if(gActionIs(AttachTo) && powerOn)
      "The vending machine starts to hum. ";
  }
;

++ vendingSlot : RestrictedContainer, Component 'small vertical slot' 'slot'
  "It's labelled <<label>>. "
  label = 'ADMITTANCE THREE FARTHINGs - EXACT CHANGE ONLY'
  validContents = [silverCoin, brassCoin]
  notifyInsert (obj, newCont)
  {
    "\^<<obj.theName>> drops into the slot and ";
    if(obj!=silverCoin || !vendingMachine.powerOn)
    {
      "immediately falls straight through into the hole. ";
      obj.moveInto(vendingHole);
      exit;
    }
    "a moment or two later a ticket appears in the hole. ";
    museumTicket.moveInto(vendingHole);
    obj.moveInto(nil);
    exit;
  }
```


TADS 3 Tour Guide

```
}  
;  
  
++ vendingHole : Container, Component 'hole' 'hole'  
    "It's a small square hole at about chest height. "  
    bulkCapacity = 1  
;  
;
```

There's not much new in the above. We define `vendingMachine.powerOn` to determine whether the power cable is attached to both the vending machine and the power socket, and we use `afterAction` to start the machine humming when both ends are first plugged in, just to confirm that the vending machine is now powered up (if we wanted to be more sophisticated we could also add a suitable `Noise` object here, although we'd also have to add suitable handling to take it away again). Perhaps the most significant method we define here is `vendingSlot.notifyInsert`, which checks both that the right coin has been inserted and that the power is on before issuing a ticket. If a ticket is issued then we move the coin into `nil` to make it no longer available, and then `exit` to prevent the normal action handling from moving the coin back into the slot.

Next we need to define the all-important power cable:

```
powerCable : PlugAttachable, Attachable, Thing  
    'thick black power cable/cord/lead/plugs' 'black cable' @galleyCupboard  
    "It's a thick black power cable, about four feet long, with  
    plugs both ends. "  
    bulk = 2  
    canAttachTo(obj) {return obj is in (socket, vendingMachine); }  
    travelWhileAttached (movedObj, traveler, connector)  
    {  
        if(movedObj==self)  
        {  
            foreach(local cur in attachedObjects)  
                tryImplicitAction(DetachFrom, self, cur);  
        }  
    }  
;  
;
```

The straightforward part is the `canAttachTo` method which simply defines the socket and the vendingMachine as the objects that the cable can be attached to. The subtler part is the `travelWhileAttached` method. This is needed to cater for the situation where the player character walks off still holding the cable while it's plugged into either the socket or the vending machine or both. What we make it do in this case is to detach the cable from whatever it's attached to; using **tryImplicitAction** to perform this decoupling makes this detachment an implicit action here, which creates quite a neat effect.

Finally, of course, we need to define that entrance ticket (which starts off without a location):

```
museumTicket : Readable 'small green ticket' 'small green ticket'  
    "It's a ticket for admission into the museum. "  
    readDesc = "Eerhtsdad Caves Museum of Curious Antiquities\n  
    ADMIT UP TO TWO PERSONS\nMULTIPLE TIMES TILL EXPIRY DATE\n    Expiry date: TBA "  
    bulk = 0  
    weight = 0  
;  
;
```

At this point you should be able to compile and run the game and try out the vending machine. Eventually we'll get to the point where you'll need the ticket to get into the museum.

14.5. PermanentAttachment

The `PermanentAttachment` class can be used for objects that are permanently attached to each other, and are described as being mutually attached. Since they're described as being attached the player may attempt to detach them, and the permanent attachment relationship basically provides a means of displaying an appropriate message in response.

One way of setting up a `PermanentAttachment` relationship is to make both the objects `PermanentAttachments` and add one to the `attachedObjects` property of the other, for example:

TADS 3 Tour Guide

```
PermanentAttachment, FireSource, Fixture 'flaming torch torch/flame/flames' 'torch' @mainCave
    "The torch, which is fixed firmly to wall by a steel bracket, is blazing merrily,
    its flames casting a bright but flickering light over the cave. "
    cannotTakeMsg = 'It\'s fixed to the wall. '
    preCondiObjBurnWith = static inherited - objHeld
    isLit = true
    disambigName = 'flaming torch'
    attachedObjects = [bracket]
    baseCannotDetachMsg = 'It\'s too firmly fixed. '
;

bracket : PermanentAttachment, Decoration 'steel bracket' 'steel bracket' @mainCave
    "The steel bracket is fixed securely to the wall; there doesn't appear to be
    any way it could be detached. "
;
```

Note that we can override **baseCannotDetachMsg** to provide a customized response.

The other way of setting up this relationship is where one object is located inside another, for example because it's a Component of the first object. If the child object is given the **PermanentAttachmentChild** class, then the attachment relationship between the two objects is set up automatically; for example:

```
PermanentAttachment, Fixture 'wooden (deck) rail' 'deck rail' @quarterDeck
    "The wooden deck rail runs along the forward edge of the Quarterdeck,
    separating it from the main deck, although it is possible to get round
    the rail either to starboard or port to go forward. A large wooden
    panel is fixed to the centre of the rail. "
    baseCannotDetachMsg = 'You can\'t; it\'s permanently fixed to the rail. '
;

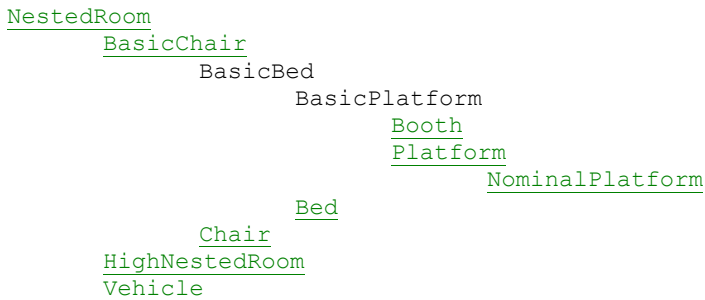
+ PermanentAttachmentChild, Component 'large wooden panel' 'panel'
    "The panel seems to have something to do with sailing the ship. A wheel and a lever
    are mounted on it, and between them is a hexagonal aperture. "
;
```

Note that with this setup, the **baseCannotDetachMsg** must be overridden on the parent object, not the child, to have any effect.

15. NestedRoom

15.1. NestedRoom Overview

A NestedRoom is an object that can contain an actor without being a fully-fledged Room. A NestedRoom is typically an object an actor can stand, sit or lie on within a room, such as a bed or chair.



Although it doesn't strictly belong in the NestedRoom category, we'll also look at the [OutOfReach](#) class here, since it doesn't fit in any other obvious grouping, and it will be convenient to consider it alongside HighNestedRoom.

15.2. NestedRoom

You're not likely to use the NestedRoom class directly in your code, but since we shall be looking at many of its subclasses, it's worth starting with this base class.

According to the comments in the library code:

A Nested Room is any object that isn't a room but which can contain an actor: chairs, beds, platforms, vehicles, and the like.

An important property of nested rooms is that they're not fully-fledged rooms for the purposes of actor arrivals and departures. Specifically, an actor moving from a room to a nested room within the room does not trigger an actor.travelTo invocation, but simply moves the actor from the containing room to the nested room. Moving from the nested room to the containing room likewise triggers no actor.travelTo invocation. The travelTo method is not applicable for intra-room travel because no TravelConnector objects are traversed in such travel; we simply move in and out of contained objects. To mitigate this loss of notification, we instead call actor.travelWithin() when moving among nested locations.

By default, an actor attempting to travel from a nested location via a directional command will simply attempt the travel as though the actor were in the enclosing location.

The point to note here that a NestedRoom is not a fully-fledged Room; it is not a TravelConnector, is not usually entered or left via TravelConnector, and does not normally employ the TravelVia method to enter or leave it (or at least, not when simply leaving the NestedRoom for its enclosing room, as in getting off a bed or chair).

To receive notification of entering or leaving a NestedRoom, you can't use enteringRoom(), leavingRoom, travelerArriving(), and travelerLeaving, but you can use **actorTravelingWithin(origin, dest)** (defined on BasicLocation, and hence available to Room as well as NestedRoom). As the comment in the library code describes it, the purpose of this method is to:

Receive notification of travel among nested rooms. When an actor moves between two locations related directly by containing (such as from a chair to the room containing the chair, or vice versa), we first call this routine on the origin of the travel, then we move the actor, then we call this same routine on the destination of the travel.

This routine is used any time an actor is moved with travelWithin(). This is not used when an actor travels between

TADS 3 Tour Guide

locations related by a *TravelConnector* object rather than by direct containment.

We do nothing by default. Locations can override this if they wish to perform any special handling during this type of travel.

Some important, or at least potentially useful, properties of *NestedRoom* (inherited by and hence common to its subclasses) include:

- **bulkCapacity** - the maximum bulk the nested room can hold - usually a large number, it can be made smaller, for example to limit the seating capacity of a chair.
- **exitDestination** - this is where an actor ends up when s/he gets out of this nested room. By default, this will simply be the nested room's location.
- **roomName** - the room name that's displayed on the status line when the player character is in this nested room and can't see out (e.g. the nested room is a trunk and the trunk is closed). By default, this is simply the nested room's name.
- **stagingLocations** - The valid "staging locations" for this nested room. This is a list of the rooms from which an actor can DIRECTLY reach this nested room; in other words, the actor will be allowed to enter 'self', with no intervening travel, if the actor is directly in any of these locations. If the list is empty, there are no valid staging locations. The point of listing staging locations is to make certain that the actor has to go through one of these locations in order to get into this nested room. This ensures that we enforce any conditions or trigger any side effects of moving through the staging locations, so that a player can't bypass a puzzle by trying to move directly from one location to another without going through the required intermediate steps. Since we always require that an actor go through one of our staging locations in order to enter this nested room, and since we carry out the travel to the staging location using implied commands (which are just ordinary commands, entered and executed automatically by the parser), we can avoid having to code any checks redundantly in both the staging locations and any other nearby locations. By default, an actor can only enter a nested room from the room's direct container. For example, if a chair is on a stage, an actor must be standing on the stage before the actor can sit on the chair.

15.3. BasicChair

It is most unlikely that you would need to use the *BasicChair* class in your game: it would represent something you could sit on but not put anything on. However, it is important to discuss some of the features of *BasicChair* that are inherited by *Chair*, *Bed* and *Platform*, and are hence common to these classes that you most likely will use.

The important properties to note on *BasicChair* are:

- **allowedPostures** - a list of the postures that are allowed on this chair (out of a selection of *sitting*, *standing*, and *lying*). For a chair this is normally *sitting* and *standing*, but you could optionally disallow standing (for example, if the chair is the back seat of a car) or allow lying (for example, if the chair is a large settee). Beds and platforms also allow lying by default, but again this can be changed.
- **obviousPostures** - a list of the *obvious* postures for this object. For a chair, this would normally be just sitting, since although you can stand on a chair, this is not the obvious posture to adopt on it.
- **defaultPosture** - the posture adopted by default for this object (in the case of a chair, this would be *sitting*).

15.4. Platform

A *Platform* is a *NestedRoom* that an actor can stand on (or also sit on or lie on, though standing on will normally be the most obvious). We'll add a museum lobby with both a carpet and a table that can be stood on:

```
museumLobby : Room 'Entrance Lobby' 'the entrance lobby'
"A welcoming sign next to the main museum entrance to the west declares:\n
<<museumSign.desc>>.\nOtherwise there's not much to this small lobby
apart from plain white walls, a battered old table, and a shabby carpet .
A brightly-lit corridor leads off to the east. "
east = brightCorridor
west = museum
;

+ museumSign : Readable, Fixture 'sign' 'sign'
"ABSOLUTELY NO ADMITTANCE\nTO THE MUSEUM\nWITHOUT A TICKET"
;
```

TADS 3 Tour Guide

```
+ Platform, Heavy 'shabby old carpet' 'shabby old carpet'
  "It may once have been red with a dark blue pattern - or perhaps
    it was the other way round. "
;

+ batteredTable : Platform, Heavy 'battered old table' 'battered old table'
  "This veteran table has clearly seen sterling service for decades,
    but the poor old thing looks like it's coming up to retirement: it's
    scratched, chipped, and generally battered, and one of the legs is
    starting to look distinctly crooked. "
  actionDobjStandOn
  {
    inherited;
    "As you stand on the table it creaks alarmingly. ";
  }
  tableHere = (isIn(museumLobby) ? 'a battered old table, ' : '')
;
```

You'll also need to add `west = museumLobby` to the definition of `brightCorridor`.

15.5. NominalPlatform

A `NominalPlatform` is a named place where an NPC can stand (or sit, or lie). This can be used to have NPCs described as standing in relation to a particular part of the room, e.g. "by the door" or "under the ceiling light". By default the NPC will be described as standing "on" the `NominalPlatform`, but this can easily be changed by defining the `actorInPrep` to something other than 'on'. For example, to allow an NPC to be described as "standing behind the table" in the museum lobby, we could add this after the room definition:

```
+ behindTable : NominalPlatform
  name = 'table'
  actorInPrep = 'behind'
;
```

Note that we do not give this `NominalPlatform` any description or `vocabWords`, since it is not an object the player is meant to interact with directly. We thus have to assign the name property explicitly.

To see this actually doing anything, we need to add an NPC; since we have not yet reached the `Actor` class, we'll make him about as minimal as we can:

```
Person 'curator' 'curator' @behindTable
  "The curator is a ferret-faced little man in his mid-forties. "
  isHim = true
;
```

When you visit the museum lobby you will now see that the curator is described as standing behind the table. At this point, though, you may be wondering whether this was really worth the bother; why not simply state this in the curator's description? The answer is that we might want the actor to move around, so that we don't always want him to be described as standing behind the table. We can demonstrate this by starting to define the museum proper and adding another `NominalPlatform` there:

```
museum : Room 'Museum of Curious Antiquities' 'the main museum'
  "Many display cases line this long chamber, each containing its own curious
    exhibit. There seems to be another chamber off to the south, marked by
    a sign saying <q>Benefactors' Exhibition</q>, while the main exit is to the east. "
  east = museumLobby
;

+ byCases : NominalPlatform
  name = 'display cases'
  actorInPrep = 'by'
;
```

To show this at work we need to move the curator from one place to the other. To do this we'll have the curator follow the player character in and out of the museum:

TADS 3 Tour Guide

```
curator : Person 'curator' 'curator' @behindTable
    "The curator is a ferret-faced little man in is mid-forties. "
    isHim = true
    beforeTravel(traveler, connector)
    {
        inherited(traveler, connector);
        if(traveler==gPlayerChar && connector is in (museum, museumLobby))
        {
            "The curator follows you <<connector==museum ? 'in' : 'out'>>. ";
            if(connector == museum)
                moveIntoForTravel (byCases);
            else
                moveIntoForTravel (behindTable);
        }
    }
;
```

If you want to have the curator standing by the display cases but sitting behind the table, you could add the line `posture = sitting` to the initial definition of the curator, and then `posture = standing`; and `posture = sitting`; after the two `moveIntoForTravel` statements, i.e.:

```
if(connector == museum)
{
    moveIntoForTravel (byCases);
    posture = standing;
}
else
{
    moveIntoForTravel (behindTable);
    posture = sitting;
}
```

Although raises the question of what the curator is sitting on, and whether one should not then implement a chair object to place him in, instead of a `NominalPlatform`.

The standard postures defined in the library are standing, sitting and lying, so if you want to have an NPC described as doing anything else in a `NominalPlatform` (such as leaning against the doorpost), you'll either have to define a new posture (which would be rather laborious for a one-off use) or else override a batch of methods on the `NominalPlatform`. According to the comments in the library code, 'For more elaborate customizations, such as "leaning against the streetlamp", you'll need to override `actorStandingHere`, `statusStanding`, `actorStandingDesc`, `listActorStanding`, and `actorStandingGroupPrefix/Suffix`'. Unfortunately the last two don't appear to be properties of `NominalPlatform`, (they're actually methods of the `libMessages` object and can hardly be customised on a per `NominalPlatform` basis). One can, however, override all the rest on a custom class:

```
class VNominalPlatform : NominalPlatform
    postureName = 'standing'
    actorInPrep = 'on'
    statusStanding(actor) { " (<<postureName>> <<actorInName>>)" ; }
    actorStandingDesc(actor)
    { "\^<<actor.itIs>> <<postureName>> <<actorInName>>. "; }
    actorStandingHere(actor)
    { "\^<<actor.nameIs>> <<postureName>> <<actorInName>>. "; }
    listActorStanding(actor)
    { "\^<<actor.nameIs>> <<postureName>> <<actorInName>>. "; }
;
```

Then if one wanted to describe the curator as "leaning against the display cases" you could simply change that `NominalPlatform` to:

```
+ byCases : VNominalPlatform
    name = 'display cases'
    actorInPrep = 'against'
    postureName = 'leaning'
;
```

And it will probably work well enough (and remove the need to set `curator.posture`). Whether this is worth the effort for a single case is debatable, but it may become worthwhile if you wanted to make a lot of use of `NominalPlatforms` describing various postures.

15.6. Bed

A Bed is simply a NestedRoom an actor can lie on (or sit on). The likely place for a bed is in a bedroom, and as we'll need to pay King Solomon a visit sometime to collect his legendary purple carbuncle (no you won't find it mentioned in the Bible, but we can always imagine that it's one of the precious stones the Queen of Sheba gave him at 2 Kings 10.10) it may as well be his bedroom.

```
solomonBedroom : Room 'Bedroom' 'the bedroom'
  "This large, square bedchamber is panelled in cedar, with a non-glazed window
    looking out to the north, and a large bed against the east wall. A door leads
    out to the west. "
  west = solBedroomDoor
  out asExit(west)
;

+ solBedroomDoor : Door 'bedroom cedar door' 'bedroom door'
  "It's made of cedar. "
;

+ Bed, Heavy 'large bed' 'large bed'
  "It has a cedar frame and an unsprung mattress. "
;

+ Fixture 'window' 'window'
  "The window is basically an opening in the north wall, comprising
    two rectangular sections each with curved tops. It is unglazed. "
  dobjFor(LookThrough)
  {
    action()
    {
      "The window overlooks a building site where a large team of workers
        are constructing a small temple. ";
    }
  }
;
```

15.7. Chair

Having provided King Solomon somewhere to sleep, we can go on to provide somewhere for him to sit - a chair in the neighbouring room (along with one or two useful items):

```
solomonStudy : Room 'Study' 'the study'
  "This is another square, cedar-panelled room, with windows to north and south
    and doors to east and west. It is largely bare apart from a table and chair
    over by the northern window. "
  east = solBedroomDoorOutside
  west = studyDoor
;

+ solBedroomDoorOutside : Door
  -> solBedroomDoor 'bedroom door*doors' 'bedroom door'
;

+ studyDoor : Door 'study door' 'study door'
  "Predictably enough, it's made of cedar. "
  checkDobjOpen
  {
    reportFailure('You\'d better not go that way. ');
    exit;
  }
;

+ solomonTable : Surface, Heavy 'table' 'table'
  "It's a plain, square, cedarwood table. "
;
```

TADS 3 Tour Guide

```
++ baarasRoot : Thing 'flame-coloured flame coloured baaras root/mandrake'
    'flame-coloured root'
    "The flame-coloured root is forked and fleshy. "
;

++ carbuncle : Thing 'purple carbuncle/ruby/stone' 'purple carbuncle'
    "It's an amazingly large stone - probably a ruby, but with a curious purple tinge. "
;

+ solomonChair : Chair, Heavy 'chair' 'chair'
    "The chair is made of cedar. It's quite large and elegantly carved with
    pomegranates, but is totally unpadded and doesn't look terribly
    comfortable. "
;
```

And just to show this works, we can put the king himself in his chair:

```
solomon : Person 'king solomon' 'King Solomon' @solomonChair
    isHim = true
    isProperName = true
    posture = sitting
;
```

Okay, so you think a study is a bit of an anachronism in a tenth-century BCE palace; you're probably right, but then where did Solomon sit around while he was getting all that wisdom? The Baaras root, by the way, will be used for its exorcistic properties. In the mean time, let's add another chair, this time in the cabin of our ship:

```
cabinChair : Chair 'padded chair/cushion' 'chair' @greatCabin
    "It's a fine wooden chair with a round back and a padded cushion. "
    initSpecialDesc = "A wooden chair sits behind the desk. "
    bulk = 10
    weight = 7
;
```

15.8. HighNestedRoom

A HighNestedPlatform is a NestedRoom that represents a part of the room that's too high to get into, unless we provide some means of reaching it. An example might be the upper of two bunk beds. The obvious place to put pair of bunk beds might be in the crew quarters of our ship. In order to reach the top bunk the player must be standing on the chair we conveniently left in the main cabin aft; to achieve this we modify the **stagingLocations()** property of the HighNestedRoom to return the chair, but only when the actor's standing on it:

```
Bed, Fixture 'bottom bunk bed*bunks*beds' 'bottom bunk' @crewQuarters
    "The bottom bunk is mounted low on the port side, under the top bunk. "
;

HighNestedRoom, Bed, Fixture 'top bunk bed*beds*bunks' 'top bunk' @crewQuarters
    "The top bunk is mounted high on the port side, above the bottom bunk. "
    stagingLocations()
    {
        local lst = new Vector(5);
        if(gActor.posture==standing)
            lst.append(cabinChair);
        return lst.toList;
    }
;

+ Thing 'pillow' 'pillow'
    "It's just a plain white pillow. "
;
```

At the same time we need to change the description of the crewQuarters room to reflect the addition of some new furniture:

TADS 3 Tour Guide

```
crewQuarters : DarkCabin 'Crew Quarters' 'the crew quarters'
  "The crew quarters seem largely deserted, apart from a single locker
  fixed to the bulkhead, and a pair of bunk beds nestling against against
  the port side. There's an exit back aft and a ladder leading down into
  the hold. Another exit leads forward. "
  ...
;
```

If you compile and run this, you'll find that you can only get onto the top bunk if you're standing on the chair, but that there's a couple of things that are less than ideal. First, if you try to get on the bunk and the chair isn't present, the game tells you that you need to be sitting on the chair first, which is both incorrect (you actually need to be standing on it) and too revealing. Even worse, if the chair is present the game sits you on the chair and then tells you that the bunk is too high. The best solution is probably to modify stagingLocations so that it only returns the chair if the actor is actually standing on it. At the same time we'll take advantage of the revision to make it easier to add further things the player could stand on to reach the top bunk:

```
HighNestedRoom, Bed, Fixture 'top bunk bed*beds*bunks' 'top bunk' @crewQuarters
  "The top bunk is mounted high on the port side, above the bottom bunk. "
  stagingLocations()
  {
    local lst = new Vector(5);
    foreach(local cur in stagingPlatforms)
      if(gActor.posture==standing && gActor.isIn(cur))
        lst.append(cur);
    return lst.toList;
  }
  stagingPlatforms = [cabinChair]
;
```

One other thing you may have noticed is that although we can only get onto the top bunk with the aid of the chair, there is nothing to stop us taking the pillow while we're standing on the floor. There's nothing unrealistic about this; indeed it could well be that we could quite easily reach up and take the pillow without being able to get ourselves onto the top bunk, it just doesn't happen to be the behaviour we want here. To prevent the pillow being reached except from the top bunk we need to go a stage further and make the latter an [OutOfReach](#).

15.9. OutOfReach

The OutOfReach class is a mix-in class that can be used to put its contents, and optionally itself, out of reach of anyone outside it. Whether the OutOfReach object is itself out of reach is determined by its **canObjReachSelf(obj)** property, which returns true if obj (normally an actor) can reach the OutOfReach object and nil otherwise. Likewise **canObjReachContents(obj)** returns true if obj (again normally an actor) can reach the OutOfReach object's contents and nil otherwise. There are two corresponding methods for reaching things from inside the OutOfReach object: **canReachFromInside(obj, dest)** returns true if obj (again normally an actor) can reach dest, and nil otherwise; and **canReachSelfFromInside(obj)** returns true if the OutOfReach can be reached by an obj (once again normally an actor) located within the OutOfReach and nil otherwise. Note that canReachSelfFromInside(obj) is defined as { return canReachFromInside(obj, self); }, which means that if you want an actor to be able to reach an OutOfReach from within itself you can do so either by overriding canReachSelfFromInside(obj) to return true, or by overriding canReachFromInside(obj, dest) to return true when dest is self (amongst other things for which you might want it to return true).

Again this is all very abstract, so let's turn our top bunk into a concrete example. Because it's already a HighNestedRoom we don't want the bunk also to be unreachable from the outside by virtue of being an OutOfReach, so we need its canObjReachSelf method to return true. It seems reasonable that an actor should be able to reach the contents of the top bunk when s/he's in a position to get onto it, so we want canObjReachContents to return true when obj is located in a valid staging location for the top bunk (if we're actually on the bunk we'll automatically be able to reach its contents). Finally, while we're on the bunk we should be able to reach the bunk itself and also anything we could use as a staging platform to reach the bunk. At the same time we'll make looking under or taking the pillow worthwhile by revealing a vital piece of paper:

TADS 3 Tour Guide

```
OutOfReach, HighNestedRoom, Bed, Fixture 'top bunk bed*beds*bunks' 'top bunk' @crewQuarters
"The top bunk is mounted high on the port side, above the bottom bunk. "
stagingLocations()
{
    local lst = new Vector(5);
    foreach(local cur in stagingPlatforms)
        if(gActor.posture==standing && gActor.isIn(cur))
            lst.append(cur);
    return lst.toList;
}
stagingPlatforms = [cabinChair]
canObjReachSelf(obj) { return true; }
canObjReachContents(obj) { return stagingLocations.indexOf(obj.location); }
canReachFromInside(obj, dest)
{ return dest==self || stagingPlatforms.indexOf(dest); }
;

+ Underside 'plain white pillow' 'pillow'
"It's just a plain white pillow. "
bulkCapacity = 1
allowPutUnder = (!location.ofKind(Actor))
;

++ leftHalfPaper : Hidden, Readable 'left half torn sheet yellow paper*sheets*papers'
'torn sheet of yellow paper'
"It looks like the left hand half of a sheet of paper that's been torn in two. It
contains a list of letters and numbers. "
readDesc = "A0\nA2\nC9\nJ8\nM3\nQ7\nT5\n"
mainExamine
{
    if(!described) name += ' (left half)';
    inherited;
}
;
```

The importance of this sheet of paper is obvious. There is no way the player character can be expected to find which of the various possible settings of the slider and dial in the Tardis will actually bring it to a useful destination, so we provide a list of the settings that do. Note that we only add ' (left half)' to the display name of this piece of paper once the player examines it to ascertain that it actually is the left half. While we're at it we might as well supply the other half - behind the picture we'll put in that cabinet aboard the Tardis:

```
OpenableContainer, Fixture 'storage cabinet' 'storage cabinet' @tardisLivingQuarters
"The large cabinet is painted a cream colour and looks securely fixed to the wall. "
;

+ smallPicture : RearSurface 'small picture/photo/photograph' 'small picture'
"It's a faded photograph of an eccentrically-dressed man with a
long scarf, in company with a smiling young woman with
long blonde hair. "
allowPutBehind = nil
;

++ rightHalfPaper : Hidden, Readable 'right half torn sheet yellow paper*sheets'
'torn sheet of yellow paper'
"It looks like the left hand half of a sheet of paper that's been torn in two. It
contains a list of names. "
readDesc = "Ship Hold\nSpaceStation\nNew Mars\nJerusalem\n
Nivalis\nLondon\nOutside Caves\n"
mainExamine
{
    if(!described) name += ' (right half)';
    inherited;
}
;
```

Meanwhile, there's one other thing we may want to customize on our OutOfReach. If (for example) you try to take the pillow without being on the top bunk or standing on the chair, you'll be told "The pillow is too far away." You may think that this makes it sound like it's a long way off, rather than fairly close at hand and just out of reach, so it may be you'd prefer "The pillow is out of reach. " This seems a better message altogether for the OutOfReach class, so we'll create a new library message and then modify OutOfReach to use it:

TADS 3 Tour Guide

```
modify playerActionMessages
  outOfReachMsg(obj)
  {
    gMessageParams(obj);
    return '{The obj/he} {is} out of reach. ';
  }
;

modify OutOfReach
  cannotReachFromOutsideMsg(dest) { return &outOfReachMsg; }
  cannotReachFromInsideMsg(dest) { return &outOfReachMsg; }
;
```

15.10. Booth

A booth is a nested room that serves as a small enclosure within a larger room. Booths can serve as regular containers as well as nested rooms, and can be made openable by addition of the Openable mix-in class. Note that booths don't have to be fully enclosed, nor do they actually have to be closable. Examples of booths might include a cardboard box large enough for an actor to stand in, a closet and a shallow pit.

For our example of a booth we'll put a large packing case in the temple cellar; the trick is that it has a secret compartment that only opens when the case itself is closed. Thus the only way for an actor to discover the secret compartment is to climb inside the packing case and close it while inside. We'll also put three gold coins in plain view in the case as a red herring, although the inscription on the coins (Greek for Vespasian) provides a (probably pretty cryptic) clue to the provenance of the ancient packing case.

```
templeCellar : DarkRoom 'Cellar beneath Temple' 'the cellar beneath the Temple'
...
;

+ packingCase : Openable, Booth, Heavy 'large packing case/lid' 'large packing case'
"The packing case, which looks large enough to get into, looks positively
ancient. It's made of some kind of wood that has darkened with age but
might possibly be cedar. "
initSpecialDesc = "An ancient wooden packing case sits in the corner. "
useInitSpecialDesc()
{
  /* show our initial description only when the actor isn't in me */
  return inherited() && !gActor.isIn(self);
}
interiorDesc = "Although it looked quite large on the outside, it feels
rather cramped on the inside. "
roomName = ('Inside ' + theName)
defaultPosture = (isOpen ? standing : lying)
makeOpen(stat)
{
  if(stat)
    callWithSenseContext(secretCompartment, sight,
      { : "Opening the case closes the secret compartment. " } );
  else
  {
    foreach (local cur in allContents())
    {
      /* if this is a standing actor, disallow closure */
      if (cur.isActor && cur.posture == standing)
      {
        /*
         * we can't close - issue a failure report and
         * terminate the command
         */
        reportFailure('{You/he} cannot close the packing case while
          someone inside it is standing up. ');
        exit;
      }
    }
  }
}
```

TADS 3 Tour Guide

```
    inherited(stat);
    if(!stat)
        callWithSenseContext(secretCompartment, sight,
            {:"Closing the case opens a secret compartment in the bottom of the case. " } );
}
/* enforce the low headroom when the box is closed */
makeStandingUp()
{
    if (isOpen)
    {
        /* we're open, so proceed as normal */
        inherited();
    }
    else
    {
        /* the box is closed, so they can't stand up */
        reportFailure('There\'s not enough room to stand up in
            the packing case while it\'s closed. ');
    }
}
basicExamine()
{
    if(gActor.isIn(self) && gActor.canSee(self) && !isOpen)
        interiorDesc;
    else
        inherited;
}
;

++ GoldCoin;
++ GoldCoin;
++ GoldCoin;

++ secretCompartment : Container, Fixture 'secret compartment' 'secret compartment'
    "The secret compartment runs the width of the case and about half its
    length. "
    isOpen = (!packingCase.isOpen)
    sightPresence = (isOpen)
;

+++ bronzeBowl : Container 'bronze bowl' 'bronze bowl'
    "It's of ancient and somewhat curious design, cast with two rows of
    pomegranates all the way round the outside. "
    bulkCapacity = 4
    bulk = 5
    weight = 5
;

class GoldCoin : Thing 'gold coin*coins' 'gold coin'
    "It looks ancient. The only writing you can make out on it looks
    like &Omicron;&Upsilon;&Epsilon;C&Pi;Alpha;C&Iota;&Alpha;&Nu&Omicron;C. "
    isEquivalent = true
;
```

If you have trouble with the Greek letters, you can just about get there using "OUECPACIANOC" using a P for a capital Pi (the C represents a capital Sigma, since this is the way a Sigma was frequently written in ancient texts). This is simply a transliteration of VESPASIANOS, Vespasian being the Roman Emperor at the time of the destruction of the Jerusalem temple (by his son Titus) in 70 CE, the use of Greek suggesting that the coins came from the eastern (Greek speaking) rather than western (Latin speaking) part of the empire.

Of more note to most TADS 3 authors will be the use of `callWithSenseContext` to ensure that the messages about the revealing of the secret compartment are only displayed when the player character is inside the case, and the code (largely borrowed from the TADS 3 sample game) to prevent closing the case while anyone's standing inside, or standing inside while the case is closed.

15.11. Vehicle

A Vehicle is a special kind of NestedRoom that can move around with actors inside. Specifically, if the player character is aboard a vehicle and the player issues a movement command (such as NORTH, IN, or GO THROUGH DOOR) it is the vehicle that moves (conveying the player character and any other passengers with it).

Note that the Vehicle class is not the only way in which to implement a vehicle, both the ship and the Tardis have shown that vehicles in the broader sense can often be implemented as one or more rooms whose exits can be manipulated to produce the desired movement effect. The Vehicle class is suited for relatively small vehicles, such as bicycles or horses, that don't provide a complete location in themselves, and that it seems reasonably natural to control with the standard movement commands. A character on a horse might reasonably regard riding a horse as a little like a faster form of walking, allowing him or her to roam over the countryside as freely as s/he would on foot. A character in a car is usually restricted to roads, and this perhaps forms a borderline case; it may be you don't want to implement the mechanics of driving in your game and you provide a reasonably comprehensive network to drive round, and in that case the car might appropriately be implemented as a Vehicle. For larger vehicles, especially those restricted in where they can travel, such as ships, aircraft and buses, it is probably better to implement the Vehicle as one or more Rooms.

Another rule of thumb to apply when deciding whether to use the Vehicle class is how, in the context of the game, it is most natural to think of the actor's location. On a bicycle or on the back of a horse, you might think of yourself as primarily on the sloping plain or the cycle path, and only secondarily on the horse or bicycle. You would expect the room name to be shown as "Sloping Plain (sitting on the horse)" rather than "Back of Horse". Conversely, aboard a ship or even a boat you might expect the room name to be "Aboard the Boat" rather than "Somewhere in Lake (sitting in the boat)". In the former case, but not the latter, you'd want to use a vehicle.

To make a Vehicle actually useful and not merely a cosmetic equivalent to walking, it ought to be able to take the player character where he would not otherwise be able to go. To show how we might do this, we'll create a snow-covered world (which you reach with the Tardis), where the combination of snow and distance make it impossible to get very far without a snowmobile. To make the snowmobile a bit more realistic, we'll make it necessary to start up its engine before it'll go anywhere. We'll start the snowmobile off in a small wooden hut:

```
insideHut : Room 'Inside the Wooden Hut' 'inside the hut'
    "This interior of this hut affords scant shelter from the biting cold wind
    outside. "
    out = snowCliff
;

+ snowMobile : Vehicle, Chair, Heavy 'snowmobile' 'snowmobile'
    "It looks vaguely like a large motorbike on sleds. There are two seats,
    one behind the other, with handlebars at the front, in the centre of
    which is mounted the starter button and a small switch. "
    specialDesc = "The two-seater snowmobile rests on the ground. "
    initSpecialDesc = "A two-seater snowmobile stands at one end of the hut. "
    useSpecialDesc = (!gPlayerChar.isIn(self))
    engineOn = nil
    travelerPreCond(conn) { return [snowEngineOn]; }
    bulkCapacity = 20
;
```

There's a few things to note about the customization here. Firstly, we include Chair and Heavy in the class list so that the player character (and a passenger) can sit on the snowmobile, but can't pick it up. Because we've used Heavy, the snowMobile wouldn't normally be listed in room descriptions, but because it will be moving around, we need it to show up; accordingly we give it a specialDesc and an initSpecialDesc. If the player character is sitting on the snowmobile, however, we don't need these descriptions, since the presence of the snowmobile will be stated in the room name (e.g. "Inside the Wooden Hut (sitting on the snowmobile)", so we override useSpecialDesc only to return true when the player character is not sitting on the snowmobile. We define a custom engineOn property that we'll come to in a moment. The main thing it will do is disallow travel on the snowmobile unless it's true. To enforce this condition we override the **travelerPreCond(conn)** method to return a custom precondition we'll define shortly.

But first, we'll define the two essential components of the snowmobile, the button and the switch that control its engine (we should also define handlebars and seats since the description of the snowmobile mentions them, but that will be left as an exercise for the reader), together with the noise it makes:

```
++ Button, Component 'starter button' 'starter button'
    dobjFor (Push)
    {
```

TADS 3 Tour Guide

```
verify()
{
    if(snowMobile.engineOn)
        illogicalNow('The snowmobile\'s engine is already running. ');
}
check()
{
    if(!starterSwitch.isOn)
        failCheck('Nothing happens. ');
}
action()
{
    "The snowmobile's engine roars into life. ";
    snowMobile.engineOn = true;
    engineNoise.makePresent();
}
}

;

++ starterSwitch : Switch, Component 'small black switch/engine' 'small switch'
    "It's a small black switch mounted at the centre of the handlebars, next
    to the button. "
    makeOn(val)
    {
        inherited(val);
        if(val==nil && snowMobile.engineOn)
        {
            "The snowmobile engine lapses into silence. ";
            snowMobile.engineOn = nil;
            engineNoise.makePresentIf(nil);
        }
    }
    verifyDobjListenTo { logicalRank(50, 'silent'); }
    soundDesc = "The engine is currently silent. "

;

++ engineNoise : PresentLater, SimpleNoise 'engine/noise' 'engine'
    "The snowmobile engine is purring quietly as it ticks over. "
    isAmbient = nil

;
```

This arrangement ensures that to start the engine you have to turn on the switch and push the button; to stop the engine you merely need to turn off the switch. We make the engineNoise a PresentLater so that it can easily be brought in or out of play according to whether the engine is on or off; we could make it more sophisticated than a SimpleNoise, but a SimpleNoise will do the job reasonably well. Note that we have overridden its isAmbient property to be nil, so that when the engine is running we'll see a description of its noise every turn. This is fair enough given the fairly limited travel that will actually be performed on the snowmobile. The other point to note here is how we handle LISTEN TO ENGINE. Since the player might legitimately try to TURN ON ENGINE or SWITCH OFF ENGINE, we add engine to the vocabWords of the starterSwitch. When the engine is off, this is the object to which LISTEN TO ENGINE will apply, so we give it a soundDesc to report that the engine is silent. But when the engine is on we'd rather that LISTEN TO ENGINE was handled by the engineNoise object, so we make starterSwitch.verifyDobjListenTo return a relatively low logicalRank (the default would be 100). This means that when the engineNoise is present, the parser will accept it as the more logical argument to handle the LISTEN TO ENGINE command, but when it is absent, the starterSwitch will handle the command and report the silence of the engine (to be sure EXAMINE ENGINE won't be handled quite so felicitously as things stands, but again, that can be left as an exercise for the reader; to do the job properly you might need a separate engine object that redirects turn on and turn off commands to the switch).

The next job is to define the precondition that stops the snowmobile from moving when its engine is not running. This is actually a bit easier than it may at first seem; basically all we need to do is to define an object of the PreCondition class and write a **checkPreCondition** method that performs the requisite check:

```
snowEngineOn : PreCondition
    checkPreCondition(obj, allowImplicit)
    {
        if(!snowMobile.engineOn)
        {
            reportFailure('The snowmobile doesn\'t budge. ');
            exit;
        }
        else if(obj.travelBarrier.indexOf(snowmobileBarrier) == nil)
```

TADS 3 Tour Guide

```
"The snowmobile emits a healthy roar from its engine as it
  sets off. ";
}
;
```

We are cheating slightly here by using the PreCondition to report the noise the snowmobile makes if it *does* move off. This isn't the official use of a precondition, but it happens to be a convenient place - perhaps about the only convenient place - to display a message describing this sound each time the snowmobile moves. Without it, the snowmobile becomes a bit ghostly - we're reported as sitting on the snowmobile each turn, but it's not made explicit that the snowmobile has conveyed us. But we don't want this message displayed if the snowmobile doesn't actually move, and there's another reason it might not move besides its engine being switched off: the player might try to take it somewhere we decide a snowmobile shouldn't go. To restrict its movements we'll be setting up a special kind of TravelBarrier called a [VehicleBarrier](#), which is what we'll be looking at next. Here we check for the presence of that VehicleBarrier among the travel barriers associated with the connector via which the snowmobile is being asked to travel, which connector is fortunately provided for us in the obj parameter of the checkPreCondition method.

15.12. VehicleBarrier

A VehicleBarrier is a special kind of [TravelBarrier](#) that allows actors to pass (effectively on foot) but (at least by default) prevents the passage of Vehicles, though this behaviour can be overridden (by overriding its **canTravelerPass(traveler)** method) to make it more selective about which kind of vehicles it will and won't allow to pass. Here, however, we simply want to override the explainTravelBarrier method to provide a more specific message:

```
snowmobileBarrier : VehicleBarrier
  explainTravelBarrier(traveler)
  {
    reportFailure('You can\'t ride the snowmobile into there. ');
  }
;
```

Since the snowmobile is the only Vehicle we're going to implement in this game, we can be sure that the explainTravelBarrier method will only ever be called for the snowmobile; otherwise, the explainTravelBarrier(traveler) method would have to test which traveler it was taking about, or else (and perhaps preferably) canTravelerPass(traveler) would need to be overridden to block only the snowmobile.

We don't want the snowmobile to be used anywhere except the snowy world on which it's found, so the first place we'll put this barrier is on the outside door of the Tardis, to prevent the snowmobile being ridden into the Tardis (you will recall that it's too heavy to carry):

```
+ tardisDoor : LockableWithKey, Door '(tardis) door' 'door'
  disambigName = 'Tardis door'
  keyList = [tardisKey]
  travelBarrier = [snowmobileBarrier]
;
```

You may recall that the purpose of the snowmobile is to carry the player character further than he can walk on foot through the snow. We can enforce this by defining a custom TravelBarrier that allows *only* the snowmobile to pass:

```
snowBarrier : TravelBarrier
  canTravelerPass(traveler) { return traveler==snowMobile; }
  explainTravelBarrier(traveler)
  {
    if(traveler == gPlayerChar)
      "You trudge for half an hour or so through the snow, but then you
      are forced to realize that you can't keep it up for long, so
      you turn round and come back. ";
  }
;
```

Note that the way we have defined it, this TravelBarrier will provide a 'soft' boundary for travelers on foot; in other words it will make any TravelConnector with which it's associated act like a FakeConnector for pedestrians but like a normal connector for those on the snowmobile. Note further that when the player character is riding the snowmobile, it's the snowmobile that's considered to be the traveler (in the sense of the traveler parameter here), even though its passengers are in a sense traveling with it.

TADS 3 Tour Guide

While we're at it, we'll provide a general custom connector to provide soft boundaries to our snowy world, as a kind of FakeConnector that provides different messages depending on whether we're traveling on foot or on the snowmobile:

```
snowWorldFakeConnector : TravelConnector
  canTravelerPass(traveler) { return nil; }
  explainTravelBarrier(traveler)
  {
    if(traveler == snowMobile)
      "You ride across the vast snowy plain for several hours, without
      finding anything of interest, so you eventually decide to
      turn round and come back. ";
    else if(traveler == gPlayerChar)
      "You walk through the snow for what seems like hours, but all
      around the unremitting plain seems unchanging, so before you
      are overcome by frostbite or hypothermia you turn back. ";
  }
;
```

We'll also define a reusable connector for returning to the spot where the Tardis materializes on this world from locations that can only be reached by riding the snowmobile:

```
snowWorldConnector : OneWayRoomConnector
  -> snowWorld
  travelBarrier = [snowBarrier]
  noteTraversal(traveler)
  {
    "You ride the snowmobile across the plain back to the Tardis. ";
  }
;
```

Armed with these special connectors and barriers we can now create the rest of our snowy world, the northern reaches of which will contain a forest in which we'll hide the last of our mysterious tablets:

```
snowWorld : OutdoorRoom 'Snow-Covered Plain' 'the snow-covered plain'
  "The bleak, snow-covered plain stretches as far as the eye can see in all directions
  under a leaden sky, the only visible feature being a tall mountain range to
  the east. "
  vocabWords = 'snow plain'
  east : OneWayRoomConnector
    {
      -> snowCliff
      noteTraversal(traveler)
      {
        if(traveler is in (gPlayerChar, snowMobile))
          "You <<traveler==snowMobile ? 'ride across' : 'trudge through'>> the snow
          to the base of the mountains. ";
      }
    }
  north : OneWayRoomConnector
    {
      -> snowForestEdge
      travelBarrier = [snowBarrier]
      noteTraversal(traveler)
      { "It roars off to the north and takes you
        sweeping across the snow-clad plain until you eventually arrive
        at the edge of a forest. "; }
    }
  west : OneWayRoomConnector
    {
      -> snowPrecipice
      travelBarrier = [snowBarrier]
      noteTraversal(traveler)
      {
        "You ride it westwards across the dazzling plain,
        until the monotony threatens to send you to sleep. Fortunately
        you are just alert enough to stop at the edge of a precipice. ";
      }
    }
  south = snowWorldFakeConnector
;
```


TADS 3 Tour Guide

```
+ Distant 'small mountain range' 'small mountain range'
  "Distances are hard to judge on this almost featureless white plain, but the
  mountains do not look far off. They seem to jut abruptly up against the sky
  without the basic courtesy of intervening foothills, suggesting that they
  were formed by a geological fault running across the plain. "
;

snowCliff : OutdoorRoom 'Under the cliff' 'under the cliff'
  "The sheer cliff of the mountain range brings the plain to an abrupt end at
  this point. A small wooden hut hugs the side of the cliff. "
  west : OneWayRoomConnector
    {
      -> snowWorld
      noteTraversal(traveler)
      {
        if(traveler is in (gPlayerChar, snowMobile))
          "You <<traveler==snowMobile ? 'ride the snowmobile' : 'walk'>> back
          to the Tardis. ";
      }
    }
  in = insideHut
  north = snowWorldFakeConnector
  south = snowWorldFakeConnector
  east : NoTravelMessage { "The sheer cliff blocks progress further south,
  and there's no way you can climb it. " }
;

+ Enterable ->insideHut 'small wooden hut' 'small wooden hut'
  "It's a shabby structure, made of thin planking knocked together clumsily.
  It hasn't weathered well, either. ";
;

snowForestEdge : OutdoorRoom 'Edge of Forest' 'the edge of the forest'
  "Although the snow-filled plain stretches to the eastern, southern and
  western horizons, directly to the north lies a pine forest thick with
  snow-covered trees. "
  south = snowWorldConnector
  east = snowWorldFakeConnector
  west = snowWorldFakeConnector
  north = snowForest
;

snowForest : OutdoorRoom 'Snow Forest Fork' 'the fork in the path'
  "This forest looks like it's been deep in winter since time began. All around
  the tall pine trees are white with frost, their branches laden with
  snow. At this point in the forest the path from the south forks to
  northeast and northwest. "
  south = snowForestEdge
  northwest = nwForestPath
  northeast = neForestPath
  travelBarrier = [snowmobileBarrier]
;

nwForestPath : OutdoorRoom 'Northwest Forest Path' 'the northwest forest path'
  "Not only does this forest been in the depths of winter since time
  began, it seems to be of interminable extent, for this path running
  roughly northwest-southeast through it seem never-ending. "
  southeast = snowForest
  northwest : FakeConnector { "You carry on down the interminable
  forest path, and still there seems no end to it. " }
;

neForestPath : OutdoorRoom 'Clearing' 'the clearing'
  "The long path northeast through the forest eventually comes to an
  end in a small clearing, where branches and pine-leaves lie in
  a tangle on the ground amidst the snow. "
  southwest = snowForest
;

+ Immovable '(pine) branches/leaves/twigs/pine-leaves' 'leaves and branches'
  "A tangle of pine twigs, branches and leaves clutters the ground, mingling
```

TADS 3 Tour Guide

```
    with the snow. "
    isPlural = true
    dobjFor(LookIn) asDobjFor(LookUnder)
    dobjFor(LookUnder)
    {
        verify() {}
        action()
        {
            if(woodenTablet.moved)
                "You find nothing else there. ";
            else
            {
                woodenTablet.discover;
                "Half-buried among the leaves and twigs is a square wooden tablet. ";
            }
        }
    }
;

+ woodenTablet : Hidden, Tablet 'wooden tablet*tablets' 'wooden tablet'
    inscription = "T D A Z P\n S H S I L\nH E R O A\nC O H E N\nW H I T E"
    initSpecialDesc = "A wooden tablet lies half-buried in the snow. "
;

snowPrecipice : OutdoorRoom 'Edge of Precipice' 'the edge of the precipice'
    "A deep fault line severs the plain here, leaving a sheer drop to the west.
    A white mist drifts below, making it hard to tell lies beyond, especially
    as it's hard to distinguish white mist from white snow. The snowy plain
    continues to east, south, and north. "
    east = snowWorldConnector
    south = snowWorldFakeConnector
    north = snowWorldFakeConnector
    west : NoTravelMessage { "That way lies a sheer drop. " }
;
```

Finally, we musn't forget to make it possible for the Tardis to reach our new snow world:

```
+ tardisDestinations : SecretFixture, PreinitObject
    destinations = static new LookupTable
    execute()
    {
        destinations['A0'] = hold;
        destinations['A2'] = spaceStation;
        destinations['C9'] = redDesert;
        destinations['T5'] = outsideCave;
        destinations['Q7'] = londonStreet;
        destinations['M3'] = snowWorld;
    }
    ...
;
```

16. MultiLocs

16.1. MultiLoc

A MultiLoc is an object that exists (or, at least, can exist) in several locations at once. We have already used several MultiLocs in the course of developing the game without much comment, examples including the mast in the [FloorlessRoom](#) example, the lake in the [Decoration](#) example, and of course the specialized subclasses of MultiLoc, the [DistanceConnector](#) and the [SenseConnector](#).

Strictly speaking, a MultiLoc (as opposed to its specialized subclasses) represents a single object that exists in several locations by virtue of being situated on their boundary, like a statue at the centre of a square where the square is divided into several locations. The mast we defined earlier is a good example of that, since it's an object standing between the port deck and the starboard deck and is thus equally in both. Note that it remains physically the same mast whichever side of the deck we climb it from or whatever we do to it (e.g. chop it down).

A second safe use for MultiLoc is for a Distant object that looks exactly the same from several locations, e.g. a distant mountain or the moon.

Although under some circumstances you can get away with using MultiLoc for other purposes (as we have up to now), for a large object that spans many locations you should normally use [MultiFaceted](#), and for a [Decoration](#) or [Fixture](#) you want cloned in many locations you should normally use [MultiInstance](#). There are two main reasons for this. The first is sensory - if a MultiLoc is visible in one location it is visible in all. Suppose, for example, that your player character is wandering around a forest at night, carrying a torch. If you created a MultiLoc to represent the trees, then if the player character dropped the torch and moved to another (dark) location, the trees would still be visible, although for everything else it would be too dark (which is not what you want). The second is that a MultiLoc represents something that remains physically the same object. If you allow the player to cut a notch in a tree and this is reflected in its description, then once one MultiLoc tree is notched, every tree in the forest will appear notched.

Of course there *can* be cases where these situations will never arise. In practice if you are creating a Decoration, Distant or other type of NonPortable with which the only possible interaction is EXAMINE and every location in which it occurs is permanently lit (or else it's an object like the moon that provides its own illumination) then no practical problems should arise from using a MultiLoc to represent the object (or objects) - but you have to be very sure that these conditions obtain and won't be changed in the course of developing the game, otherwise you should use MultiInstance or MultiFaceted. And of course, there are cases, such as the fountain at the centre of the square or the mast between two halves of the deck where MultiLoc is precisely what you do want (whatever may befall it), because it represents the same physical object.

MultiLoc is a mix-in class, and should thus always be used with (and before) a Thing derived class, e.g. (note that this example shouldn't be added to the game):

```
statue : MultiLoc 'statue' 'statue'
    "It's a large stone statue of Jeremiah Foamwheeze. "
    locationList = [seSquare, swSquare, neSquare, nwSquare]
;
```

Even if the square is in darkness and you leave your torch in the southeast corner of the square, it's not unreasonable the torch would continue to illuminate the statue as seen from the other three corners of the square, and again, if you stick your hat on top of the statue while standing in nwSquare there's no reason why you shouldn't be able to retrieve it again while standing in swSquare.

Note that we don't set the location property of a MultiLoc; instead we provide a list of locations in its **locationList** property, or else use one of the other methods to initialize its locations that are described in connexion with [MultiInstance](#) below. Again, note that locationList is used only to *initialize* the list of locations a MultiLoc starts in at the beginning of the game. You cannot subsequently move a MultiLoc around by directly manipulating this list (e.g. by writing statements like `locationList += forestPath;` or `locationList = [forestPath, lakeside];`). Instead you have to use the methods **moveInto**, **moveIntoAdd**, and **moveOutOf**. These will be described in more detail under MultiInstance (where the same thing applies).

16.2. MultiInstance

The MultiInstance class is used for creating copies of objects, typically Decorations, in multiple locations. This can often be preferable to using [MultiLoc](#) for this purpose, since a MultiLoc is a single object, and thus (for example) will be lit for all locations even if it's lit only in one.

A typical use for a MultiInstance might be for creating trees in a forest. The trees are not interestingly distinctive, but they are numerically distinct objects. To put a 'trees' Decoration object in each location where we mention the forest, we first create a MultiInstance shell object and set its **locationList** property to the list of locations we want the trees to appear in, and then set up a template trees object in its **instanceObject** property.

```
MultiInstance
  locationList = [snowForest, nwForestPath, neForestPath, snowForestEdge]
  instanceObject: Decoration { 'pine tree/trees/pines' 'trees'
    "The trees, mainly tall pines, are covered in snow. "
    isPlural = true
  }
;
```

This is all you have to do; the library does the rest of the work by creating an instance of your template object in each of the rooms listed in the locationList. The library also adds MultiInstance to the class list of the objects it creates (so you don't have to bother). This basically handles specialized handling for **moveInto** so you can move the instance objects around (assuming you can find a way to refer to them) and everything will be kept in sync with the MultiInstance object. Probably the more useful way to relocate MultiInstance objects dynamically is to call the **moveInto(loc)**, **moveIntoAdd(loc)**, and **moveOutOf(loc)** methods of the MultiInstance object. For example, if we wanted the forest to grow into a new location during the course of the game (which we don't), we could give the MultiInstance object a name (such as 'trees') and then call:

```
trees.moveIntoAdd(newLocation);
```

Note that if we called `trees.moveInto(newLocation)` the trees would be removed from every other location. If on the other hand we wanted to chop down part of the forest (which we don't) we could call:

```
trees.moveOutOf(neForestPath);
```

Finally, we don't have to use the locationList to specify the initial location of the trees. As an alternative, we could have defined a ForestRoom class for use at each of our forested locations, and then have used MultiInstance's (or rather MultiLoc's) **initialLocationClass** property to define where we wanted the trees to appear:

```
MultiInstance
  initialLocationClass = ForestRoom
  instanceObject: Decoration { 'pine tree/trees/pines' 'trees'
    "The trees, mainly tall pines, are covered in snow. "
    isPlural = true
  }
;
```

As a third (though in this case utterly pointless) alternative, we could simply have overridden the **isInitiallyIn(obj)** method:

```
MultiInstance
  isInitiallyIn(obj) { return obj in in (snowForest, nwForestPath, neForestPath,
snowForestEdge); }
  instanceObject: Decoration { 'pine tree/trees/pines' 'trees'
    "The trees, mainly tall pines, are covered in snow. "
    isPlural = true
  }
;
```

There may, however, be more complex cases in which this would be useful for implementing a rule for determining where the MultiInstanceInstances should start out. Lastly, and even more pointlessly in this case, you could override **buildLocationList()** to construct the initial list of locations:

```
MultiInstance
  buildLocationList()
```

TADS 3 Tour Guide

```
{
    local lst;
    lst = new Vector(16);
    lst.append(snowForest);
    lst.append(nwForestPath);
    lst.append(neForestPath);
    lst.append(snowForestEdge);
    return lst.toList();
}
instanceObject: Decoration { 'pine tree/trees/pines' 'trees'
    "The trees, mainly tall pines, are covered in snow. "
    isPlural = true
}
;
```

Of course the method could simply have been defined:

```
buildLocationList() { return [snowForest, nwForestPath, neForestPath, snowForestEdge]; }
```

But since there is absolutely no point in overriding the method in this case anyway, we might as well make it appear more complicated to simulate a situation in which you *might* actually want to use this method; though it's hard to think of a case where this could achieve something that couldn't be achieved just as well by overriding `isInitiallyIn(obj)`.

16.3. MultiFaceted

MultiFaceted is almost identical to [MultiInstance](#), except that it is used to place copies of what is notionally the *same* object in multiple locations, rather than copies of what are similar but numerically distinct objects. For example, although the forest contains many trees, it is only one forest, so that if we might want to refer to both the trees and the forest throughout its extent we should use a [MultiInstance](#) to represent the trees and a MultiFaceted to represent the forest:

```
MultiFaceted
locationList = [snowForest, nwForestPath, neForestPath, snowForestEdge]
instanceObject : Fixture {'frosty forest' 'forest'
    "The frosty forest is full of tall snow-laden pines growing close together. "
}
;
```

The only real difference here is that the objects set up as instances by the MultiFaceted class have MultiFacetedFacet instead of MultiInstanceInstance as their mix-in class (again this is all handled by the library so you don't have to worry about it), which in turn means that the instances are regarded as facets of one another. The *practical* upshot of this is that you can EXAMINE FOREST in one location, go to any other location in which the forest exists, and type EXAMINE IT and the parser will know that you're still talking about the forest, whereas if you EXAMINE TREES in one location, the parser won't know what you're talking about if you immediately try to EXAMINE THEM in another.

Apart from this difference, everything said about [MultiInstance](#) applies to MultiFaceted and need not be repeated. You can use exactly the same methods to move MultiFaceted objects around dynamically, or specify the initial locations the instance objects should occupy.

Some time back we defined a lake decoration object thus:

```
MultiLoc, Decoration 'great (giant) underground lake/water' 'lake'
    "The lake looks as flat as a millpond. "
    locationList = [lakeRoom, pathEnd]
;
```

This is actually harmless in this case, but to be on the safe side we could make it a MultiFaceted, and we could even take advantage of the change to vary the description of the lake depending on where it's being viewed from:

```
MultiFaceted
locationList = [lakeRoom, pathEnd, starboardDeck, westShore, eastShore, southShore]
instanceObject : Decoration
{
    'great (giant) underground lake/water' 'lake'
    "The lake, which stretches as far <<whichDirection>> as you can
```

TADS 3 Tour Guide

```
    see, looks almost as flat as a millpond, although the occasional
    ripple runs across its surface. It is also strikingly
    phosphorescent, casting an eerie green glow over the whole
    vast cavern. "
    whichDirection = (miParent.whichDirection(location))
}
whichDirection(loc)
{
    switch(loc)
    {
        case lakeRoom :
        case pathEnd: return 'south';
        case westShore : return 'east';
        case southShore : return 'north';
        case eastShore : return 'west';
        case starboardDeck : return 'to starboard';
        default : return 'away';
    }
}
;
```

17. Collections

17.1. CollectiveGroup (static)

A `CollectiveGroup` is useful when you want some actions, typically `EXAMINE`, to work in a summary fashion rather than on every object that might match. In the abstract this probably doesn't mean a great deal, so let's create a specific example. A little while back we created a museum. The natural thing to do is to put some exhibits in it:

```

museum : Room 'Museum of Curious Antiquities' 'the main museum'
  "Many display cases line this long chamber, each containing its own curious
  exhibit. There seems to be another chamber off to the south, marked by
  a sign saying <q>Benefactors' Exhibition</q>, while the main exit is to the east. "
  east = museumLobby
  south = benefactorsRoom
;

+ Decoration 'sign' 'sign'
  "<FONT COLOR=BLUE>BENEFACTORS' EXHIBITION</FONT>\n
  Special Exhibits reserved for our benefactors\n
  Sorry! No bags allowed (we have to be careful!)"
;

+ Decoration 'amber amulet/amazement*cases exhibits' 'Amber Amulet of Amazement'
  "The pendant on display in this case is indisputably amber, and quite possibly
  an amulet. The accompanying plaque explains that its principal source of amazement
  is that it performed no useful function whatsoever despite having several times
  changed hands at ever more inflated prices. <i>Quam credulus emptor nemo ineptior</i>
  the plaque concludes pompously. "
  aName = (theName)
  collectiveGroups = [exhibitGroup]
;

+ Decoration 'green gargoyle/gloom*cases exhibits' 'Green Gargolye of Gloom'
  "The Green Gargoyle of Gloom is so ugly it would probably turn the Gorgon Medusa
  to stone. The small plaque affixed to the display case explains that it was made
  by the mad demons of Hell Fire Cavern in a fit of psychotic fury, and then cast
  out into the world of men as being too hideous for mere demonkind to bear. It
  subsequently caused a public scandal when King Freddie the Fatuous mistook it
  for a bust of his wife."
  aName = (theName)
  collectiveGroups = [exhibitGroup]
;

+ Decoration 'lost king crown/peregrine*cases exhibits' 'Lost Crown of King Peregrine'
  "Frankly, this does not seem to be the most impressive exhibit in the room. The
  Lost Crown of King Peregrine the Pipsqueak looks like it came out of a
  cheap Christmas cracker and then got itself coated with even cheaper tinsel.
  The plaque on the display case nevertheless assures all who care to read it
  that this was indeed genuinely the crown of this most ignominious of
  insignificant monarchs. "
  aName = (theName)
  collectiveGroups = [exhibitGroup]
;

+ Decoration 'invisible naked mantle/emperor*cases exhibits' 'Invisible Mantle of the Naked
Emperor'
  "The display case housing the Invisible Mantle appears to be empty, but the plaque
  affixed to it declares that the Naked Emperor's Invisible Mantle is indeed there,
  woven of the finest invisible cloth - so precious that none but the Naked Emperor
  dare be seen wearing it in public (and even then not around small boys who might
  ask awkward questions). "
  aName = (theName)
  collectiveGroups = [exhibitGroup]
;

```

At this stage, if the player types `EXAMINE EXHIBITS` or `X CASES` all four exhibits will be listed, together with their full descriptions. In place of this information overload, we may prefer to give the player a summary description of the

TADS 3 Tour Guide

exhibits on display; we can do this by defining a `CollectiveGroup` object with the same *plural* vocabulary that we've given to each of the exhibits:

```
+ exhibitGroup : CollectiveGroup, SecretFixture 'display *cases*exhibits' 'exhibits'
  "The exhibits include such rare curiosities as the Amber Amulet of Amazement, the
    Green Gargoyle of Gloom, the Lost Crown of King Peregrine the Pipsqueak
    and the Invisible Mantle of the Naked Emperor. "
;
```

By default, straight out of the box, the `CollectiveGroup` only handles the `EXAMINE` command, but if we want we can change this by overriding `isCollectiveAction(action, whichObj)` and providing suitable handling on the `CollectiveGroup`. For example, we might want a collective response to an attempt to `TAKE EXHIBITS`:

```
+ exhibitGroup : CollectiveGroup, SecretFixture 'display *cases*exhibits' 'exhibits'
  "The exhibits include such rare curiosities as the Amber Amulet of Amazement, the
    Green Gargoyle of Gloom, the Lost Crown of King Peregrine the Pipsqueak
    and the Invisible Mantle of the Naked Emperor. "
  isCollectiveAction(action, whichObj)
  {
    /* we handle 'Examine' and 'Take' */
    if (action.ofKind(ExamineAction) || action.ofKind(TakeAction))
      return true;

    /* it's not one of ours */
    return nil;
  }
  dobjFor(Take)
  {
    verify() { illogical('The exhibits are all fixed in place and protected
      by an alarm system. '); }
  }
;
```

(It may be debatable whether information about a previously unmentioned alarm system belongs in a call to `illogical`, but it should be reasonably obvious to the player that s/he can't just walk off with museum exhibits, so this seems the best place to put it; besides, there's no way we could put the 'fixed in place' part of the message in `verify()` and the 'alarm system' part in `check()` and have both displayed).

If you wanted, you could take this a stage further still and have `exhibitGroup.isCollectiveAction` simply return `true`; any attempt to do anything to the exhibits will then give a perfectly good summary response. In the meantime, we should add the Benefactors' Room referred to in the south property of the museum, together with the Golden Banana of Discord in its specially protected display case:

```
benefactorsRoom : Room 'Benefactors\' Exhibition Room' 'the benefactors\' room'
  "This room seems brighter and cleaner than the main section of the museum,
    but is curiously bare, apart from a single glass display case in the middle
    of the room. "
  north = museum
  out asExit(north)
;
```

```
+ bananaCase : Container, Fixture 'glass display case' 'display case'
  "The case is of much higher quality than the display cases outside; standing on
    a polished oak plinth the large square case is held together with polished
    brass strips framing the glass sides and top. A polished brass plaque fastened
    near the top of the plinth declares the contents of the case to be the Golden
    Banana of Discord. "
  isOpen = nil
  material = glass
  dobjFor(CutWith)
  {
    verify()
    {
      if(isOpen) illogicalNow('It\'s already been cut open. ');
    }
    check()
    {
      if(gIobj != diamondRing)
        failCheck('{You/he} can\'t cut it with that. ');
    }
    action()
  }
;
```


TADS 3 Tour Guide

```
{
    "{You/he} cut{s} open the glass display case. ";
    if(gActor != gPlayerChar)
        "<q>There you are!</q> {you/she} declare{s}, <q>Easy!</q>\<.p>";
    isOpen = true;
    microphone.notifyCut();
}
}
;

++ goldenBanana : Thing 'golden banana/discord' 'Golden Banana of Discord'
    "It's about the shape and size of an ordinary banana, but seems to be made
    of solid gold. "
    aName = (theName)
    weight = 6
    isListedInContents = (!isIn(bananaCase))
;

+ Fixture 'polished oak plinth' 'plinth'
    "The polished oak plinth supporting the glass display case is about a foot square
    and three feet tall. It's quite plain in design, apart from a decoratively
    carved slat just beneath the case. "
;

++ Component 'decoratively carved leaf slat/pattern' 'ridge'
    "The slat, decoratively carved with a leaf pattern, runs round the top
    of the plinth, just beneath the glass display case. Close inspection
    reveals a small round black disc inset into one side of the slat, just above
    the plaque. "
;

+++ microphone : Attachable, Component 'small round black disc/disk/microphone'
    'small round disc'
    "Closer examination reveals that it could very well be a small
    microphone. <<rename>>"
    rename { name = 'microphone'; }
    canAttachTo(obj) { return obj==stickingPlaster; }
    handleAttach(other)
    { other.moveInto(location.location); }

    moveWhileAttached(movedObj, newCont)
    {
        if(movedObj != self && newCont != location.location)
            tryImplicitAction(DetachFrom, movedObj, self);
    }
    isMajorItemFor(obj) { return obj==stickingPlaster; }
    notifyCut()
    {
        if(!isAttachedTo(stickingPlaster))
        {
            curator.moveIntoForTravel(benefactorsRoom);
            "Almost at once an alarm bell starts ringing and a couple
            of seconds later {the curator/he} comes running into the
            room to find out what's happened. He stares aghast at the
            opened case.<.p>";
        }
    }
}
;

++ bananaPlaque : Component 'brass plaque' 'brass plaque'
    "The plaque attached to the front of the case explains that this
    indeed <i>The</i> famous Golden
    Banana, forged aeons ago in the fires of Mount Gloom, and borne out
    of Hell Fire Cavern by Benedict the Banana-Bearer at the cost of his
    sanity and his left big toe-nail. The plaque also warns:\b
    <q>This artefact is the most highly-treasured property of the state:\n
    Anyone attempting to steal it will be terminated without trial!</q> "
;


```

As you'll observe, the intention is to allow the player character or Sarah to cut open the glass case with the diamond ring, but to have the sound of their doing so picked up by a microphone that triggers an alarm unless the player has first attached a sticking plaster over the microphone to prevent it. For this to work we need to make sure that the

TADS 3 Tour Guide

sticking plaster, defined long ago as part of the contents of the first aid box, is also an Attachable:

```
++ stickingPlaster : Attachable, Thing 'sticking adhesive plaster' 'sticking plaster'
    "It's a small, round flesh-coloured sticking plaster that might just cover
      a small cut or blister. "
;
```

Note: as from TADS 3.0.10 it became possible for an object to be associated with more than one CollectiveGroup. Previously it was possible for an object to belong to only one collectiveGroup at a time, and membership of that group would have been indicated by setting the old collectiveGroup property to the single collectiveGroup object thus:

```
+ Decoration 'amber amulet/amazement*cases exhibits' 'Amber Amulet of Amazement'
    "The pendant on display in this case is indisputably amber, and quite possibly
      an amulet. The accompanying plaque explains that its principal source of amazement
      is that it performed no useful function whatsoever despite having several times
      changed hands at ever more inflated prices. <i>Quam credulus emptor nemo ineptior</i>
      the plaque concludes pompously. "
    aName = (theName)
    collectiveGroup = exhibitGroup
;
```

This way of doing things is still legal, and it will still work, but it is now regarded as 'obsolescent', so that new game code should instead use the collectiveGroups property (with an s on the end) set to a list of groups (even if the list contains only a single member) as in the previous examples. The practical reason for avoiding the now obsolescent collectiveGroup property in new game code is that support for it may eventually be withdrawn.

17.2. CollectiveGroup (mobile)

The previous example may have seemed a little complicated, but at least the CollectiveGroup and its members all stay in the same place. What if we want a CollectiveGroup that represents a collections of portable objects that can move around during the course of the game?

A case in point might be the two halves of the torn yellow piece of paper. At some point the player might want to be able effectively to join the two halves together and read the two columns side by side, instead of one after the other. With a bit of thought and effort, we can bring this about with a CollectiveGroup, such that when both halves of the piece of paper are available the player can command X YELLOW SHEETS or READ YELLOW SHEETS and see the paper whole.

The first step is to define the collectiveGroup object we want to use on both halves of the piece of paper:

```
++ leftHalfPaper : Hidden, Readable 'left half torn sheet yellow paper*sheets*papers'
    'torn sheet of yellow paper'
    "It looks like the left hand half of a sheet of paper that's been torn in two. It
      contains a list of letters and numbers. "
    readDesc = "A0\nA2\nC9\nJ8\nM3\nQ7\nT5\n"
    mainExamine
    {
        if(!described) name += ' (left half)';
        inherited;
    }
    collectiveGroups = [yellowPaperGroup]
;

/* Recall this is in a different location */

+ smallPicture : RearSurface 'small picture/photo/photograph' 'small picture'
    "It's a faded photograph of an eccentrically-dressed man with a
      long scarf, in company with a smiling young woman with
      long blonde hair. "
    allowPutBehind = nil
;

++ rightHalfPaper : Hidden, Readable 'right half torn sheet yellow paper*sheets*papers'
    'torn sheet of yellow paper'
```

TADS 3 Tour Guide

```
"It looks like the right hand half of a sheet of paper that's been torn in two. It
contains a list of names. "
readDesc = "Ship Hold\nSpaceStation\nNew Mars\nJerusalem\n
    Nivalis\nLondon\nOutside Caves\n"
mainExamine
{
    if(!described) name += ' (right half)';
    inherited;
}
collectiveGroups = [yellowPaperGroup]
;
```

The next task is to define the yellowPaperGroup CollectiveGroup object (to which we give no location):

```
yellowPaperGroup : CollectiveGroup 'yellow paper *sheets' 'sheets of yellow paper'
    "When you hold the two sheets of paper together, you find that the combined sheet
    reads:\nA0 Ship Hold\nA2 Space Station\nC9 New Mars\n
    J8 Jerusalem\nM3 Nivalis\nQ7 London\nT5 Outside Caves\n"
    dobjFor(Examine)
    {
        verify() {}
        preCond = [leftSheetHeld, rightSheetHeld]
    }
    basicExamine
    {
        if(gActor.isLocationLit)
            desc;
        else
            desc;
    }
    dobjFor(Read) asDobjFor(Examine)
    isCollectiveAction(action, whichObj)
    {
        /* we handle 'Examine' & 'Read'*/
        if (action.ofKind(ExamineAction) || action.ofKind(ReadAction))
            return true;

        /* it's not one of ours */
        return nil;
    }
    ;
```

What we have done here is, first of all, define the description we want displayed when the two halves of the piece of paper are read or examined together, and then provided a handling for the Examine command that displays this description. We also include ReadAction among the kinds of action yellowPaperGroup will handle by overriding its isCollectiveAction method appropriately, and redirecting READ to EXAMINE. The main complication is that we don't want the player to be able to EXAMINE or READ both halves of the sheet of paper together unless they are both available to him or her separately. We therefore make it a precondition of examining the sheets together that they are both held (although we will allow them to be picked up in an implicit action if they are available to be picked up). We finally need to define the custom preconditions; since they'll be so similar we'll define a custom class and then derive both the preconditions from it:

```
class SheetHeldPreCondition : PreCondition
    sheet = nil
    checkPreCondition(obj, allowImplicit)
    {
        /* check to see if the actor is holding the sheet - if so, we're done */
        if (sheet.isHeldBy(gActor))
            return nil;

        /* the actor isn't holding the sheet, try a "take sheet" command */
        if (allowImplicit && tryImplicitAction(Take, sheet))
        {
            /*
             * make sure that leaves the actor holding the sheet - if not,
             * exit silently, since the reason for failure will have
             * been reported by the "take sheet" action
             */
            if (!sheet.isHeldBy(gActor))
                exit;
        }
    }
;
```

TADS 3 Tour Guide

```
        /* indicate that we executed an implicit command */
        return true;
    }

    /* we can't take the sheet implicitly - report the problem and exit */
    reportFailure('You don\'t have both sheets. ');
    exit;
}

;

leftSheetHeld : SheetHeldPreCondition
    sheet = leftHalfPaper
;

rightSheetHeld : SheetHeldPreCondition
    sheet = rightHalfPaper
;
```

18. Scripts

18.1. Script

The basic Script class provides the skeletal framework for a simple state machine. You're more likely to use one of the subclasses of Script (namely [EventList](#) or one of its subclasses) than Script itself, though there may be occasions when Script itself is useful.

The interface provided by the Script class is rudimentary, to say the least. It provides a **curScriptState** property (which is 0 by default), a **getScriptState()** method which simply returns **curScriptState**, and a **doScript** method which does nothing at all.

The kind of situation in which this basic framework might be useful is when you want something to happen at irregular or widely spaced intervals: for example, suppose we'd like an occasional atmospheric message to be displayed in the graveyard. If we get the roomDaemon to call the Room's doScript method we could do something like this:

```
graveyard : Room, Script 'Graveyard' 'the graveyard'
  "There is something decidedly eerie about this underground graveyard with its
  musty old tombstones. This is truly a place of death; nothing
  lives here, for this place never sees the sun; a dusty path leads off to the
  northeast and a strange, stone temple is situated just to the west. "
  northeast = westShore
  west = temple
  roomDaemon
  {
    if(!ghost.isIn(self)) doScript;
    if(!ghost.moved && !statue.isPulled)
    {
      ghost.moveInto(self);
      ghost.initiateConversation(ghostTalking, ghostNode);
      ghostAppearingEvent.triggerEvent(ghost);
    }
  }
  doScript()
  {
    switch(curScriptState++)
    {
      case 1: msg1; break;
      case 3: msg2; break;
      case 6: msg1; break;
      case 10: msg2; break;
      case 15: curScriptState = 0;
    }
  }
  msg1 = "There's a sound like a distant moaning. "
  msg2 = "Something seems to flicker, but perhaps it's only a
  trick of the light. ";
;
```

In this case, it would probably be easier to put the two message strings in the Room's atmosphereList and use one of the EventList classes to control the frequency of display. There may be cases, however, where being able to control exactly when things occur like this is what you need.

18.2. EventList

An EventList is a [Script](#) object that works through a list of events in sequence, until the list is exhausted, after which it does nothing. Unlike a bare Script, which supplies only the bare framework, EventList is a fully-functional class, although one may often want to use one of its subclasses.

Normally all one needs to specify on an EventList (and many of its subclasses) is its eventList property. The **doScript** method of the EventList automatically steps through each event in sequence. But what, in this context, is an event? It

TADS 3 Tour Guide

may be either a single-quoted string, a function pointer, another Script object, a property pointer, or nil. To expand on this we may start by quoting the comment in the library code:

The script is driven by a list of values; each value represents one step of the script. Each value can be a single-quoted string, in which case the string is simply displayed; a function pointer, in which case the function is invoked without arguments; another Script object, in which case the object's doScript() method is invoked; a property pointer, in which case the property of 'self' (the EventList object) is invoked with no arguments; or nil, in which case nothing happens.

This base type of event list runs through the list once, in order, and then simply stops doing anything once we pass the last event.

This may be clearer if we give an example of an EventList using each type of entry in turn.

```
sampleEventList : EventList
/* The EventList template allows us to specify the eventList property as the first property
after the
* class name without needing explicitly to specify eventList =
*/
[
    'First we display this string. ',

    new Function { "Then we do whatever we define inside this function. ";
                  someOtherObject.someOtherMethod;
                  "Which can be more or less what we like. "; },

    otherScript, // call otherScript.doScript

    &anotherProperty, // execute the code in self.otherProperty;

    nil // finally, do nothing (which makes this element superfluous here)
]
anotherProperty { "This can also do whatever we want it to. "; }
;

otherScript : EventList
[
    'This will be displayed as the first element of otherScript. ',
    'And this will be shown as the second element, if otherScript.doScript is ever called
again. '
]
;
```

In the case of a function that uses only a single statement, we can use the short-form anonymous function syntax. This can be useful, for example, for using double-quoted strings in an EventList:

```
anotherEventList : EventList
[
    { : "<q>Hi, my name's <<sarah.makeProperName()>>.</q> she tells you. " },

    '<q>You already know my name.</q> Sarah reminds you. ',

    { : "<q>Oh - my last name's Smith.</q> she explains.
        <<gPlayerChar.setKnowsAbout(smithName)>> " }
]
;
```

An EventList with a single entry can sometimes be useful, as a way of having something happen once only. For this type of use, see the examples under [CyclicEventList](#) and [InitiateTopic](#).

18.3. StopEventList

A StopEventList is just like an [EventList](#), except that when the last item in the list is reached, it is then repeated indefinitely.

This is probably most useful in providing sequential responses in NPC conversations (which we'll be coming to in due course). For example, if you repeatedly ASK SARAH ABOUT HERSELF, you'd expect her not to give the same answer each time. Since, however, you cannot actually provide her with an infinite number of responses, you could use a StopEventList to provide, say, two or three different ones which progressively reveal more information, and a final one (to be then repeated indefinitely), which indicates that this particular topic has now been exhausted, while perhaps reminding the player of the essentials of what's been said.

We'll be meeting plenty of examples of this in connection with the conversation system, for example in conjunction with [GiveShowTopic](#), [AskTopic](#), [TellTopic](#), and [AskTellTopic](#).

18.4. CyclicEventList

A CyclicEventList is just like an EventList, except that it keeps looping round the list; once it reaches the last item in the list it goes back to the first, and carries on round indefinitely.

Although one would normally want to randomize the events that occur (or strings that are displayed), there may be occasions when you want more control over the sequence of events. For example, when our intrepid adventurer finally reaches the side of the volcano, he'll have to wait a couple of turns until the volcano happens to set fire to a nearby bush before the bush can be pulled up revealing a small hole into the side of the volcano. We could control this by means of a CyclicEventList linked to a StopEventList that sets fire to the bush first time round and merely displays some text thereafter. The following code shows how we might go about this, at the same time providing several other examples of EventLists, an example of a [SenseDaemon](#), and a means of disposing of the pesky Golden Banana of Discord:

```
sideOfVolcano : OutdoorRoom 'Side of Volcano' 'the side of the volcano'
  "Halfway up the side of the volcano the path up from the basalt plain
  peters out. There's no obvious way to climb any further up from here,
  and the fact that the volcano is continuously belching fire, smoke,
  ash, and the occasional rock and dollop of lava probably would
  probably make any further progress up its side more than a little
  perilous. "
  down = volcanoPathDown
  in = bushHole
  atmosphereList : CyclicEventList
  {
    [
      'A loud rumbling comes from deep inside the mountain. ',
      lexicalParent.rockFallList,
      nil
    ]
  }

  rockFallList : StopEventList
  {
    [
      'High above your head a shower of flame and sparks issues from the
      top of the volcano, sending blazing rocks tumbling down the
      hillside. ',
      new function
      {
        "A blazing rock comes hurtling down the hillside, narrowly
        missing you but setting the bush ablaze. ";
        bush.daemonID = new SenseDaemon(bush, &burnDaemon, 1, bush, sight);
      },
      'A shower of sparks and rocks spews from the volcano above your head. '
    ]
  }
;
```

TADS 3 Tour Guide

```
+ bush: CustomImmovable 'dessicated burning bush' 'dessicated bush'
"The bush is <<daemonID == nil ? 'little more than a collections of
dried sticks, with only the occasional apology for a leaf doing duty
for foliage' : 'ablaze'>>. "
inRoomDesc = "The only sign of life on this barren hillside
is a dessicated bush. "
burnDaemon { eventList.doScript; }
daemonID = nil
eventList : EventList
{
[
'The bush is alight. ',
'The bush is burning furiously. ',
'The bush is starting to burn out. ',
&swap
]
swap()
{
"The flames on the bush die out, leaving only charred remains. ";
lexicalParent.daemonID.removeEvent();
lexicalParent.daemonID = nil;
burnedBush.moveTo(lexicalParent.location);
lexicalParent.moveTo(nil);
}
}
cannotTakeMsg = 'Tug as {you/he} will, the bush will not not quite come
free of the ground. '
;

+ volcanoPathDown : StairwayDown 'narrow down path' 'narrow down path'
;

+ bushHole : HiddenDoor 'small hole' 'small hole'
"The small hole exposed by pulling up the burned-out bush looks
just large enough to crawl through. "
destination = volcanoTunnel
isOpen = (burnedBush.isPulled)
;

burnedBush : Lever, Fixture 'burned bush' 'burned bush'
"All that remains are the charred remains of the bush. "
makePulled(stat)
{
if(stat)
{
"When {you/he} pull{s} at the burned-out bush, it comes clear of the
ground, sending a shower of dried earth tumbling down the hillside
and revealing a small hole in the hillside. ";
}
else
{
"You push the remains of the bush back into the hole. ";
}
inherited(stat);
}
;

volcanoTunnel : DarkRoom 'Narrow Tunnel' 'the narrow tunnel'
"This narrow tunnel is only just about large enough to crawl
through. One way it carries on north deep into the volcano, while
the other way (south) leads back out. "
out asExit(south)
in asExit(north)
north = volcanoLedge
south = sideOfVolcano
;

volcanoLedge : Room 'Ledge inside Volcano' 'the ledge inside the Volcano'
"This narrow ledge, on the inside of the volcanic crater, overlooks a
heaving pool of lava below. It's very hot here and probably not a
healthy place to be, but the only way out is through a small hole
```


TADS 3 Tour Guide

```
in the side of the crater. "
in = volcanoTunnel
south asExit(in)
atmosphereList : EventList
{
  [ nil,
    'A plume of lava shoots up from the pool, throwing rocks out
      of the crater mouth above. ',
    nil,
    'A second plume of lava bubbles up from the pool, narrowly
      missing the ledge. ',
    'Yet another jet of steam, lava, rocks and ashes shoots up
      from the pool, uncomfortably close. ',
    { "This time, the hissing lava bubbling up from the lake
        strikes the ledge, scalding you to death.\b<<endGame(ftDeath)>>" }
  ]
}
enteringRoom(traveler)
{
  atmosphereList.curScriptState = 1;
}
;

+ Enterable ->volcanoTunnel 'small hole' 'small hole'
"The small hole in the crater wall is just about large enough to
crawl through and looks like the only way to leave the ledge. "
;

+ Distant 'lava pool' 'lava pool'
"The pool of seething red lava is bubbling away about two hundred yards
beneath the ledge, accordingly throwing up flaming rocks and spurts
of lava. "
iobjFor(ThrowAt) asIobjFor(ThrowTo)
iobjFor(ThrowTo)
{
  verify() {}
}
throwTargetCatch(obj, path)
{
  "^<<obj.theName>> drops into the bubbling pool of lava and disappears
out of sight, lost forever. ";
  obj.moveTo(nil);
}
;
```

Note, that at the moment, there's no way to reach the sideOfVolcano room to try all this out other than using POW DESSICATED BUSH if you've included ncDebugActions with your code.

18.5. RandomEventList

A RandomEventList is an EventList that randomly selects one of its items each turn its doScript method is run. In practice, the [ShuffledEventList](#), which inherits from RandomEventList, may be more useful, since it ensures that the same event will not be chosen twice in succession. The typical use of a RandomEventList might be in the atmosphereList of a [Room](#) or combined with a [TopicEntry](#) to provide a series of random conversational responses.

RandomEventList defines, (and hence ShuffledEventList inherits) three useful properties to control the frequency with which random messages are displayed (or random events occur). This is principally designed for use in a Room's atmosphereList (or perhaps an Actor's 'fidget list'), where once players have seen all the messages once or twice, they've got the general idea and may start to find them unduly repetitious.

You can set the **eventPercent** property (an integer between 0 and 100) to define the proportion of turns in which you want one of the events (normally just displaying a string) to occur (for example, setting eventPercent = 75 would cause a random event to be selected on average in three turns out of every four). By default, eventPercent is 100. Additionally, if you wish, you can have this percentage fall (or even rise!) to a new value, **eventReduceTo** after **eventReduceAfter** turns. If **eventReduceAfter** is nil (as it is by default) then the eventPercent never changes. This behaviour is in fact provided by [RandomFiringScript](#).

18.6. ShuffledEventList

The Shuffled event list is just like a random event list, except that it fires its events in a "shuffled" order rather than an independently random order. "Shuffled order" means that events are fired in random order; once an event is fired it is not fired again until all of the other events have been fired. The effect is similar to dealing from a pack (UK) / deck (USA) of cards.

For the first time through the main list, the strings are normally shuffled immediately at startup, but this is optional. If **shuffleFirst** is set to nil, the events will *not* be shuffled the first time through; they'll be run through once in the order listed, then shuffled for the next time through, then shuffled again for the next, and so on. So, if you want a specific order for the first time through, just define the list in the desired order and set shuffleFirst to nil.

You can optionally specify a separate list of one-time-only sequential strings in the property **firstEvents**. This list of events will be run through once, much like a standard [EventList](#). Once they've been exhausted, the ShuffledEventList will switch to the main eventList list, showing it one time through in its given order, then shuffling it and running through it again, and so on. The firstEvents list is never shuffled - it's always shown in exactly the order given. This combination of a list in a fixed order followed by a list which gets shuffled is particularly useful in connexion with [TopicEntries](#), when you want an NPC to give a specific sequence of responses (probably conveying relatively important information), followed by a series of randomly-ordered responses (perhaps indicating that the NPC has nothing further to say on the subject).

We saw an example of a ShuffledEventList right at the start in the definition of our first [OutdoorRoom](#), where we used one for the atmosphere list. We'll also be encountering examples in connection with [AskTopic](#), [AskForAboutTopic](#), [DefaultAskTopic](#), [DefaultTellTopic](#), [DefaultAskTellTopic](#), and most of the other DefaultTopics.

The ShuffledEventList has its own [template](#) to allow easy definition of its firstEvents and eventList properties.

Since ShuffledEventList inherits from [RandomFiringScript](#), it can also use the same eventPercent, eventReduceTo, and eventReduceAfter properties that were described on [RandomEventList](#).

18.7. ExternalEventList

An "external" event list is one whose state is driven externally to the script. Specifically, the state is *not* advanced by invoking the script; the state is advanced exclusively by some external process (for example, by a daemon that invokes the event list's **advanceState()** method). In other words, each time you call the doScript() method of an ExternalEventList, the same event will fire (or the same string display), until you make an explicit call to advanceState().

A possible use for this would be for a list of strings/events where what happens/is displayed depends on some external event. Before that event occurs, you see the message/event relating to the earlier state of affairs; when the event occurs it can call the ExternalEventList's advanceState() method thereby causing the next event to be repeatedly fired, until another call to advanceState moves it on to the next, and so on.

18.8. SyncEventList

A synchronized event list is an event list that takes its actions from a separate event list object. It gets its current state from the other list, and advancing our state advances the other list's state in lock step. Set **masterObject** to refer to the master list whose state we synchronize with.

This can be useful, for example, when you have messages that reflect two different points of view on the same events: the messages for each point of view can be kept in a separate list, but the one list can be a slave of the other to ensure that the two lists are based on a common state.

18.9. RandomFiringScript

RandomFiringScript is a mix-in class that can be added to any Script-like object to reduce the percentage of time it fires. [ShuffledEventList](#) and [RandomEventList](#) already inherit from RandomFiringScript, but you can mix with any of the other Script subclasses, provided you list it first, e.g.

```
myScript: RandomFiringScript, EventList
[
    'The golden banana emits a cloud of purple smoke. ',
    ...
]
eventPercent = 50
;
```

You can set the **eventPercent** property (an integer between 0 and 100) to define the proportion of turns in which you want one of the events (normally just displaying a string) to occur (for example, setting eventPercent = 75 would cause a random event to be selected on average in three turns out of every four). By default, eventPercent is 100. Additionally, if you wish, you can have this percentage fall (or even rise!) to a new value, **eventReduceTo** after **eventReduceAfter** turns. If **eventReduceAfter** is nil (as it is by default) then the eventPercent never changes.

19. Actors & NPCs

19.1. Overview - Actors & NPCs

A Non-Player Character (NPC) is, roughly speaking, any other animate being the player/player character encounters in the game world. "Animate" may be a slightly fuzzy concept, since it might include not only other people and various animals, but also, perhaps, robots and talking computers. Ultimately it is, of course, up to game authors what they implement as NPCs; the choice will normally depend on which entities in their games exhibit sufficiently complex behaviour.

Implementing an NPC can be a complex task, but TADS 3 breaks it down into a series of (in the main) fairly simple tasks by spreading the implementation of each NPC over (potentially) a wide range of objects. The basic NPC will be defined in an object of the Actor class, or one of its subclasses, `UntakeableActor` or `Person`. The difference between these three classes is that an Actor is portable whereas the other two are not; you would therefore normally use Actor for a small animal such as a mouse or a cat that you might want the player character to be able to pick up and carry around. Neither `UntakeableActor` nor `Person` can be picked up (they both derive from [Immovable](#) as well as from Actor), the only difference is the text of the messages displayed when the attempt to pick them up or otherwise push them around is made: `UntakeableActor` displays more generic messages suitable for large animals, while `Person` displays messages more suited to the attempt to cart off a fellow human being. Thus you would normally use `UntakeableActor` for larger animals such as tigers and elephants, and `Person` for human or human-like characters. In this game all the NPCs we shall be implementing will use the `Person` class (which is not to say that they'll all be strictly human!).

Particularly for a complex NPC such as another human character, the Actor (or Person) object should generally contain only the very basic, constant data needed to define that actor (such as name, gender, vocabWords, basic description, and maybe weight, bulk and one or two other things). The behaviour of the NPC should be defined using a number of [ActorState](#), [TopicEntry](#) and perhaps [AgendaItem](#), [ConvNode](#) and [TopicGroup](#) objects nested inside the principal Actor object. In particular we use `TopicEntry` objects to define what happens in conversation with the actor (the responses to ASK ABOUT, TELL ABOUT, and ASK FOR, and also to GIVE and SHOW commands). One of the aims of this structure is to eliminate, or at least drastically reduce, the need for a huge tangle of if, else and switch statement spaghetti to define complex behaviour, although assuming your NPCs do anything out of the library-defined ordinary, the time will come when you may need the occasional if or switch statement.

As always, this may not mean a great deal in the abstract; hopefully it will all become clearer as we work through the implementation of a number of NPCs, feature by feature. In the meantime, if you have not already done so, you may wish to read the extensive article on Creating Dynamic Actors in TADS 3 in the *Technical Manual*.

19.2. Basic Actors

Without further ado, we'll start defining the basic Actor (actually Person) objects we'll be needing in this game. Because, with all their associated helper objects (`ActorStates`, `TopicEntries`, `ConvNodes`, `TopicGroups` and `AgendaItems`) actor definitions can end up getting long and complicated, it is generally worth defining each but the very simplest NPC in its own source file. We'll begin the Sarah, the young lady who will become the player character's companion and helper:

```
sarah : Person 'attractive young woman/brunette' 'young woman' @lakeRoom
    "She's an attractive brunette, somewhere in her mid-twenties. "
    isHer = true
    properName = 'Sarah'
    globalParamName = 'sarah'
;
```

Often, this is all that is needed in the definition of the basic Actor object. We define `isHer = true` so that the parser knows this NPC is female (and can be referred to with feminine pronouns such as 'she' and 'her'). Adding a **globalParamName** to an actor is nearly always useful, particularly in a case like this where the woman will change from 'the young woman' to 'Sarah' once we get to know her; it allows us to use whatever we have defined as the `globalParamName` in parameterised strings like "{The sarah/she} smiles at you warmly" and have the game display "The young woman smiles at you warmly" or "Sarah smiles at you warmly" as appropriate to our knowledge of her name. The other property we have defined, **properName**, is not a library property at all, but one we are using for our own purposes to store the name the young woman will eventually be known by. To make this useful we need to add a

TADS 3 Tour Guide

corresponding custom method to the actor class:

```
modify Actor
makeProper()
{
    if(properName != nil)
    {
        name = properName;
        initializeVocabWith(properName);
        isProperName = true;
    }
    return name;
}
;
```

What the makeProper method does is (provided properName is non nil) change the name property to the properName (e.g. "young woman" to "Sarah"), add properName to the vocabulary words for the actor (so henceforth we can also refer to the young woman as "sarah" in commands), and sets the library property **isProperName** to true so that the game knows that the actor's name is now a proper name and will display messages like "Sarah is here" rather than "The Sarah is here". Finally the method returns the name of the actor as it now is; this means it can be used in conversational text such as "<q>Hello, I'm <<sarah.makeProper>>.</q> the young woman introduces herself" both to display the name and to update the Actor object appropriately at the same time. Unfortunately, we'll have to wait till we get into programming conversations to see this in action (unless you want to add some temporary code to try it out, e.g. by making it a response in sarah.actionDobjKiss).

We have already made a temporary definition of the curator, but let's move him to his own file and set him up similarly:

```
curator : Person 'curator' 'curator' @behindTable
"The curator is a ferret-faced little man in is mid-forties. "
isHim = true
properName = 'Professor Altmeister'
globalParamName = 'curator'
;
```

Note that we have stripped out the code that moves him around, as we'll be implementing it differently. Obviously, for the male curator we set **isHim** = true. Once again we add a globalParamName.

The other character we've already met is King Solomon; here he is as we'll now create him:

```
solomon : Person 'middle-aged middle aged man/king' 'middle-aged man' @solomonChair
"He's quite good-looking in a middle-eastern sort of way, with long curly
black hair that's just starting to go grey, and a neatly kept beard. He's
dressed in a purple cloak. "
isHim = true
posture = sitting
properName = 'King Solomon'
globalParamName = 'solomon'
;
```

We can also use a single Person object to represent a group of NPCs, especially when there's no particular need to distinguish one from another:

```
demons : Person 'bunch crowd gaggle demons' 'demons' @hellPath
"They're an ugly bunch of mis-shapen, semi-substantial beings, slightly
green in tinge with glaring red eyes and tiny forked tails. "
isPlural = true
globalParamName = 'demons'
;
```

Since we've put the demons in a location we haven't created yet, we'd better add this new location:

```
hellPath : OutdoorRoom 'Path down Hell Fire Cavern' 'the path'
"About halfway down the track from the summit to the basalt plain below the track
flattens to a small ledge. Stone steps continue upwards towards the top and down
to the bleak basalt plain below, while across the plain to the north Mount Gloom
continues to belch smoke and flame. "
up = hellPathUp
down = hellPathDown
;
```

TADS 3 Tour Guide

```
+ hellPathUp : StairwayUp ->roughStaircase 'stone upward up steps' 'upward steps'
  "It looks a rough ascent, possibly treacherous in places, but probably passable
  with care. "
  isPlural = true
;

+ hellPathDown : StairwayDown 'stone downward down steps' 'downward steps'
  "The rough steps downwards to the basalt plain look even less inviting than those
  leading up, but they appear to be just about negotiable. "
  isPlural = true
;
```

You might also want to add hellPath to the locationList property of the MultiLoc, Distant volcano object defined earlier.

While we're in the business of the inhuman NPC, here's our final actor:

```
ghost : Person 'ghost' 'ghost'
  "It's as insubstantial as you'd expect a ghost to be, a pale white shape you can
  almost see through, with just a hint of washed-out colour. The wan face bears
  a once-regal demeanour, and its features vaguely resemble those on the golden
  statue in the grotto. "
  isHim = true
  isIt = true
  properName = 'Benedict'
  globalParamName = 'ghost'
;
```

We have deliberately not given this ghost a location, since it will only appear (in the cemetery) after the gold statue has been toppled (which is why we can be sure it's safe to refer to the statue in the description). Note that we can make the ghost both a "him" and an "it" by setting both isHim and isIt to true, since players may think of the ghost as being either neuter or masculine (or may change from one to the other once the ghost introduces himself as Benedict).

Five NPCs may constitute a fairly sparse population for the size of game world we are implementing, but they will suffice for the plot (such as it is) and to illustrate how NPCs can be implemented. Over the course of the next several sections we shall gradually bring them to life.

19.3. Basic Actor Customization

One of the most basic ways we can start to customize an actor is to override some of his/her standard message properties to customize the response the actor gives to such commands as KISS SARAH or HIT SARAH. We can do this very simply by defining appropriate message properties on the actor, such as **cannotKissActorMsg** and **uselessToAttackMsg**, and then assigning them a single-quoted string to contain the message we want displayed in response to KISS or HIT (for example).

These message properties will also work perfectly well if they're defined as methods that return a single-quoted string. This means we could define one of these properties, cannotKissActorMsg, say, to return one of a list of messages. In this case we might like to use a [ShuffledEventList](#), but this won't quite do, since we need something that *returns* a single-quoted string, not something that displays one. Fortunately, there is a class that can do this, namely the **ShuffledList**. We can place a list of strings (or anything else we like) in a ShuffledList's **valueList** property, and then call its **getNextValue** method to return a random element from the list. The advantage of using this method over simply using the rand() function with a list of strings is the same as the advantage of using a ShuffledEventList over a RandomEventList, namely that the ShuffledList will keep working through the complete list of items in its valueList property before repeating any of them.

Taking advantage of the message properties and a ShuffledList we could accordingly customize Sarah to cope with being hit and kissed in her own particular way:

```
sarah : Person 'attractive young woman/brunette' 'young woman' @lakeRoom
  "She's an attractive brunette, somewhere in her mid-twenties. "
  isHer = true
  properName = 'Sarah'
  globalParamName = 'sarah'
```

TADS 3 Tour Guide

```
uselessToAttackMsg = '<q>Ouch! What did you do that for?</q> she complains. '
cannotKissActorMsg { return noKissMessages.getNextValue; }

noKissMessages : ShuffledList
{
  valueList =
  [
    '<q>Hey! What do you think you\'re doing!</q> she complains. ',
    '<q>Stop that!</q> she tells you. ',
    '<q>Keep your hands off me!</q> she demands. ',
    '<q>Don\'t <i>do</i> that!</q> {the sarah/she} tells you. '
  ]
}
```

19.4. Actor Knowledge

In order to be able to talk about something, an Actor must know about it, or at least know of its existence. To test whether an actor has seen something you can use its **hasSeen(obj)** method; likewise, to test whether an actor knows about something test its **knowsAbout(obj)** method. If an actor has seen something, the actor is assumed to know about it, but the converse is not necessarily true (the actor may have heard about it without actually having seen it). The library keeps track of what objects the player character has seen automatically, but in general this can be set for an actor by calling the actor's **setHasSeen(obj)** method. When an actor learns about something by other means, you can call its **setKnowsAbout(obj)** method. Since it is the player character whose knowledge you mostly want to keep track of, the library defines a macro **gSetKnown(obj)** which is effectively an abbreviation for **gPlayerChar.setKnowsAbout(obj)**. Thus, when the player character hears about something for the first time in conversation, or by reading about it somewhere, or by any other means (other than actually seeing it), you will probably want to call **gSetKnown(obj)** to record the fact. Only when the player character knows about something can it be the topic of conversation with an NPC (for example there will be no useful response to ASK SARAH ABOUT RING until the player character knows of the existence of a ring to be asked about).

By default the library keeps track of what objects have been seen and which are known about via the **seen** and **isKnown** properties on the objects (and topics) concerned. This effectively means that only the player character's knowledge is being tracked, since **hasSeen(obj)** and **knowsAbout(obj)** will return the same whatever actor they're called on, and calling **setHasSeen(obj)** or **setKnowsAbout(obj)** on one actor sets these properties for all actors. The reason for this is that most games will only be interested in tracking what the player character has seen and knows about. The library does, however, provide support for tracking the sight and knowledge of individual NPCs if required. This is done by specifying **knownProp** and **seenProp** on the actor in question. By default these are defined as **&isKnown** and **&seen** on all actors (i.e. as pointers to the **isKnown** and **seen** properties of objects). They can, however, be set to point to new properties defined to keep track of a particular actor's knowledge.

Suppose that in addition to keeping track of what the player character knows and has seen, we want to do the same for Sarah. We can simply devise two new properties that we'll call **sarahKnows** and **sarahHasSeen** to do the job for us, and set Sarah's **knownProp** and **seenProp** to point to them. Then we can use **sarah.setHasSeen(obj)**, **sarah.hasSeen(obj)**, **sarah.setKnowsAbout(obj)** and **sarah.knowsAbout(obj)** independently of the knowledge base of the player character and all other NPCs:

```
sarah : Person 'attractive young woman/brunette' 'young woman' @lakeRoom
  "She's an attractive brunette, somewhere in her mid-twenties. "
  isHer = true
  properName = 'Sarah'
  globalParamName = 'sarah'
  knownProp = &sarahKnows
  seenProp = &sarahHasSeen
```

Having defined these properties, we could also use them to define Sarah's initial state of knowledge. If Sarah started the game knowing about the curator or having seen the golden banana (which she hasn't), for example, we could define **curator.sarahKnows = true** and **goldenBanana.sarahHasSeen = true**. Likewise, if you wanted the player character to start off with knowledge of the goldenBanana, you could set **goldenBanana.isKnown = true**.

The point of all this will become clearer with a specific example we'll be implementing when we come to look at **TopicEntries**. If the player character enters the graveyard after pushing over the statue in the Golden Grotto, the ghost will appear. Before that point neither the player character (PC) nor any NPC has any reason to suppose that the ghost exists, so there's no reason for it to be a topic of conversation. The ghost thus only becomes available as a topic of

TADS 3 Tour Guide

conversation to the PC once the PC has encountered the ghost. When the ghost appears it will charge the player character with the task of finding and destroying the golden banana. Up to that point the PC may not have heard of the golden banana, and so would not be able to converse about it. After then we'd want to mark the golden banana as known about but not seen by the PC, which would allow the PC to raise the golden banana as a topic of conversation.

Now let's consider what happens when the PC decides to discuss the ghost with Sarah. Clearly, the PC can't do this until he knows about the ghost. Also, it makes no sense for the PC to ask Sarah about the ghost until Sarah has at least heard of the ghost. Sarah can learn of the ghost by two different methods; either she was with the PC when the ghost appeared and saw it for herself, or she wasn't but learned about it from the PC's subsequent account. Sarah is pretty sceptical about ghosts, and she doesn't take this one seriously unless she sees it herself, so her responses will be quite different depending whether she's actually seen the ghost or only been told about it.

The library will take care of ensuring that the ghost cannot be raised as a topic of conversation until the PC knows about it. It's up to the author to test for Sarah's state of knowledge of the ghost and have her respond accordingly. We'll need to test whether Sarah knows about the ghost at all before allowing the player to ASK SARAH ABOUT GHOST. If she does know about the ghost we'll want to provide quite different responses depending on whether she's also seen it. Once the player character knows about the ghost he can TELL SARAH ABOUT GHOST, but again we'll want to test for `sarah.hasSeen(ghost)` to decide how she responds. Also, if this is the first Sarah has heard of the ghost we'll want to call `sarah.setKnowsAbout(ghost)`. Finally, we'll want to cater for the special case where the ghost is actually present while the PC is discussing it with Sarah.

19.5. Moving Actors Around

Since Actors are meant to model living beings, there's no reason to assume they'll always stay in the same place. That means that we need some means of moving them around the game world.

The basic rule when moving actors around in code is *never* use `moveInto(dest)`. Use **`moveIntoForTravel(dest)`** instead, or `travelTo(destination, connector, backConnector)` or `scriptedTravelTo(dest)`; or else `nestedActorAction(bob, North)` or `nestedActorAction(TravelVia, redDoor)`.

For a fuller discussion, see Mike Roberts's technical article on [NPC Travel in TADS 3](#).

It's also possible to have one actor automatically follow another (normally, but not necessarily, the player character) by putting it in an [AccompanyingState](#). This is one of the ActorState classes, which is what we'll be looking at next.

19.6. Actor States

19.6.1. Overview - Actor States

Over the course of a game a sophisticated NPC may be doing many different things - talking with the player character (or resolutely ignoring him), leaning against a wall waiting for something to happen, attacking the PC in a fit of fury, going about his or her daily business, interfering with the PC's plans or following him around to lend a helping hand, and any number of other things that authors may dream up. Where these different activities (or inactivities) are sufficiently different that we should want to describe the NPC in a different way and to have the NPC respond or behave in a different way, we can use a different ActorState. An ActorState is an abstract object (not a representation of a physical game object) which we nest inside its associated actor, and which defines the state the actor is in.

Although one can use the ActorState class itself (and in some cases, particularly with non-human NPCs this may be appropriate), often it is the more specialized subclasses of ActorState that are employed. The simplest of these, the [HermitActorState](#), is used when you want the NPC to ignore the PC altogether (because the NPC is unconscious, preoccupied with something else, or just pointedly ignoring the PC). An NPC who's ready to engage in conversation with the PC but is not actually talking with him at the moment would probably be in a [ConversationReadyState](#), which would switch to an associate [InConversationState](#) while conversation is actually in progress. For a sidekick NPC who's following the PC around of his travels (or a hostile one that's pursuing him) you'd use an [AccompanyingState](#) (together with an [AccompanyingInTravelState](#)), while if you want the NPC to lead the PC around, you might use [GuidedTourState](#) and [GuidedInTravelState](#).

TADS 3 Tour Guide

Although some of these states have special methods and properties that apply to them, they all derive a great many properties and methods in common from ActorState. The subset of these that you will most often want to override with your own code are:

- **stateDesc** - this is a message that's added to the actor's basic description (the "npcDesc" property in the Actor object). Most actors will have a permanent description that never changes - a basic description of their physical appearance - along with some extra information that describes what they're doing right now. The stateDesc lets you add this extra state-dependent part.
- **specialDesc** - displays the actor's in-room description. This is the description displayed in the room description (for example, when entering a room, or in response to a LOOK command). By default, we'll invoke the actor's actorHereDesc method. You may or may not want to call this *inherited* behaviour when overriding specialDesc; for example, quite a serviceable specialDesc can often be set up by defining it as {inherited; stateDesc; }.
- **obeyCommand(issuingActor, action)** - determines if the NPC should obey the command from the given actor to carry out the given action. By default this returns nil to refuse all commands.
- **takeTurn()** - this is called once per turn. This allows the actor to carry out scripted behaviour appropriate to the current state. Note that if you override this method you should *always* call *inherited* in your override (unless you are absolutely sure you don't want the inherited behaviour), since by default this method carries out several important steps that you don't want to disable. One of the things takeTurn does is call doScript if the ActorState also inherits from Script, which means that (for example) you can add an EventList class to the class list and takeTurn will automatically work through the eventList you define for the ActorState.
- **afterAction()** - this method can contain code to be run after the PC performs an action in the NPC's scope.
- **beforeAction()** - this method can contain code to be run before the PC performs an action in the NPC's scope; the action can be vetoed with an exit command.
- **afterTravel(traveler, connector)**
- **beforeTravel(traveler, connector)** - these methods are invoked just before and after a traveler travels via travelerTravelTo(); beforeTravel is called on each object connected by containment in its old location, and afterTravel is called on each object connected by containment to the traveler in the new location. These notifications are more precise than using beforeAction() and afterAction() with the TravelVia pseudo-action, because these methods are called only when travel is actually occurring, whereas TravelVia will fire notifications even when travel isn't actually possible. A beforeTravel() method can veto the travel action using "exit". The notification is invoked before the travel is actually performed, and even before a description of the departure is produced. Although ActorAction.beforeTravel(traveler, connector) does nothing by default, this is not so of many of its subclasses, so it's a good idea to get into the habit of calling inherited(traveler, connector) when overriding this method.
- **isInitState** - if set to true, then the library will automatically set the corresponding actor's curState property to point to that ActorState during pre-initialization; in other words, set this to true on the ActorState you want this actor to start off in.
- **arrivingTurn()** - when group travel is performed using the AccompanyingInTravelState class, this is essentially called in lieu of the regular takeTurn() method on the state that is coming into effect after the group travel; in other words, if an NPC is following the PC around, this method will be called each time they arrive at a new location.
- **autoSuggest** - set to nil to disable automatic topic inventory listing on TALK TO commands.
- **getActor()** - returns the actor object to which this ActorState belongs; you *never* want to override this method, but you may frequently want access to its result.

In addition there's a couple of properties of Actor associated with ActorStates that you need to know about: **curState** contains the current ActorState for the Actor, while **setCurState(state)** changes the Actor into the new state ActorState. To change the current ActorState of an Actor, *always* call setCurState(state), *never* set curState directly (the reason being that setCurState looks after all the necessary side-effects of the state change).

To give some idea of how this might look in practice, we could define an initial ActorState for the ghost like this:

```
+ ghostHovering : ActorState
  specialDesc = "The ghost is hovering over its tomb, staring at you intently. "
  stateDesc = "It's hovering over its tomb. "
  isInitState = true
;
```

This should, of course, immediately follow the definition of the ghost actor object.

19.6.2. HermitActorState

The HermitActorState is used for when an actor is too preoccupied, or is otherwise unable or unwilling to response to conversational approaches from the PC (such as TALK TO, ASK ABOUT, or SHOW TO). When the PC addresses any such commands to an Actor in a HermitActorState, the message defined in the HermitActorState's **noResponse** property is displayed.

None of the actors we have defined so far have any cause to enter a HermitActor state in any situation we have provided so far, but to show the principle we'll add a (somewhat tongue-in-cheek) permanently unresponsive actor:

```
skeleton : Actor, Heavy 'bleached skeleton' 'skeleton' @redRavine
    "The bones have long since been bleached white, but it's undoubtedly
    a human skeleton, of a person who was once about 5'9\" tall. "
;

+ HermitActorState
    specialDesc = "A bleached skeleton lies on the ground, one arm outstretched
    towards the south. "
    noResponse = "The skeleton is a bit too dead to attend to you right now. "
    initState = true
;
```

If you wanted something really macabre, you could add more actor states to the skeleton object so that at some point it comes to life - but one ghost in a game is probably enough. Instead, we'll give another example of a temporary HermitActorState in the next section.

19.6.3. AccompanyingState

If you want an NPC to follow your player character around, you need to put that NPC into an AccompanyingState, when by default he or she will then follow any actor wherever that actor goes. In practice you may want to limit this over-eager behaviour, which you can do by overriding the **accompanyTravel(leadActor, conn)** method to return true only for the combinations of leadActor and connector that you want the NPC to follow; typically, for an NPC who's following the PC around, you might override this method to `{ return leadActor==gPlayerChar; }` but you could apply further qualifications if, for example, there were certain connectors you didn't want this NPC to follow your PC through. The other method distinctive to AccompanyState is **getAccompanyingTravelState(leadActor, conn)**, which returns the associated [AccompanyingInTravelState](#) (which we'll explain shortly). AccompanyingState also inherits all the [ActorState](#) methods we've already seen, not least **arrivingTurn()**.

The NPC who's been elected to follow the PC on his travels in this game is the young woman Sarah. She's lost her diamond ring and won't follow the PC around until he's given it to her, so we'll put temporary handling on the sarah object to switch her into her AccompanyingState when she's given the ring (note that this isn't how we'd normally do it; here it's a temporary expedient until we come to GiveTopic).

```
sarah : Person 'attractive young woman/brunette' 'young woman' @lakeRoom
    "She's an attractive brunette, somewhere in her mid-twenties. "
    isHer = true
    properName = 'Sarah'
    globalParamName = 'sarah'

/* The next section of code is NOT how we'd do it for real -
 * As we'll see this is MUCH better handled using a GiveTopic,
 * but since we haven't got to GiveTopic yet we have to use
 * the bad old way.
 */
iobjFor (GiveTo)
{
    verify() { }
    check()
    {
        if(gDobj != diamondRing) inherited;
    }
    action()
```

TADS 3 Tour Guide

```
{
  if(gDobj == diamondRing)
  {
    "<q>Great! I've been looking for that!</q> she declares, <q>Since
    you seem to know what you're doing, I think I'll stick with you
    from now on.</q><.p>";
    gDobj.moveTo(sarah);
    setCurState(sarahFollowing);
  }
}
;

+ sarahFollowing : AccompanyingState
  specialDesc = "{The sarah/she} is standing beside you. "
  stateDesc = "She's standing beside you. "
  accompanyTravel(leadActor, conn)
    { return leadActor == gPlayerChar; }
;
```

If you try this out you'll find it works - after a fashion. Sarah will follow you around (in a strangely silent kind of way) so long as you're moving from room to room, but won't follow you into a nestedRoom. This may be fine, but does mean that, for example, if you get on the snowMobile and ride off, Sarah won't follow you. Even, if Sarah *had* followed the PC onto the snowmobile, she'd still be described as "standing beside you", whereas at that point we need her to be described as "sitting behind you." To fix the first set of problems we need a different ActorState for Sarah when she's on the snowmobile, and an appropriate test for making Sarah get on and off the snowmobile and change states. Here's how we might do it with the tools we've seen so far:

```
+ sarahFollowing : AccompanyingState
  specialDesc = "{The sarah/she} is standing beside you. "
  stateDesc = "She's standing beside you. "
  accompanyTravel(leadActor, conn)
    { return leadActor == gPlayerChar; }
  afterAction()
  {
    if(gPlayerChar.isIn(snowMobile))
    {
      getActor.setCurState(sarahOnSnowMobile);
      nestedActorAction(getActor, SitOn, snowMobile);
      "{The sarah/she} gets on the snowmobile behind you. ";
    }
  }
;

+ sarahOnSnowMobile : HermitActorState
  specialDesc = "{The sarah/she} is sitting behind you. "
  afterAction()
  {
    if(!gPlayerChar.isIn(snowMobile))
    {
      getActor.setCurState(sarahFollowing);
      nestedActorAction(getActor, GetOutOf, snowMobile);
      "{The sarah/she} climbs off the snowmobile after you. ";
    }
  }
  noResponse = "<q>Let's just get moving, shall we?</q> she suggests.<.p>"
;
```

Strictly speaking this isn't how we'd normally use a HermitActorState, since the noResponse message actually is a response; in this case it's a convenient shortcut, and we haven't encountered DefaultAnyTopic yet.

19.6.4. AccompanyingInTravelState

You'll have noticed that when Sarah follows you around, you see a message "The young woman comes with you." This is a decent enough default message for the purpose, but if you wanted to change it, here's how.

TADS 3 Tour Guide

First, we need to define a custom class derived from `AccompanyingInTravelState`, then we need to override its **`sayDeparting(conn)`** method to display whatever message we want in place of "The young woman comes with you. " We can also also override the **`specialDesc`** property to display whatever we'd like shown in the room description to describe Sarah's presence on the arriving turn. We could, for example, take advantage of this to vary randomly the message displayed each time in an effort to make it look a little less repetitive and mechanical:

```
class SarahInTravelState : AccompanyingInTravelState
  sayDeparting(conn) { departMessages.doScript; }
  specialDesc { arriveMessages.doScript; }
  departMessages : ShuffledEventList
  { ['{The sarah/she} comes with you. ',
    '{The sarah/she} follows you. ',
    '{The sarah/she} trails faithfully along behind. ',
    '{The sarah/she} follows hot on your heels. ',
    '{The sarah/she} walks along beside you. ',
    '{The sarah/she} accompanies you.'
    ] }
  arriveMessages : ShuffledEventList
  { [ '{The sarah/she} is at your side. ',
    '{The sarah/she} is still with you. ',
    '{The sarah/she} is standing close by. ',
    '{The sarah/she} takes a quick look around. ',
    '{The sarah/she} looks at you expectantly. '
    ] }
;
```

For this to do anything, we need to change the `AccompanyingState` to make use of this new `AccompanyingInTravel` state by overriding its **`getAccompanyingTravelState(leadActor, conn)`** method:

```
+ sarahFollowing : AccompanyingState
  specialDesc = "{The sarah/she} is standing beside you. "
  stateDesc = "She's standing beside you. "
  ...
  getAccompanyingTravelState(leadActor, conn)
  { return new SarahInTravelState(location, leadActor, self); }
;
```

Note that the third parameter (self) in this new `SarahInTravelState` call is the `ActorState` to which the Actor will return on completion of travel; normally we want this to be the `AccompanyingState` (hence 'self') that set the `AccompanyingInTravelState` up, but we could, if special circumstances warranted it, insert some other `ActorState` here if we wanted the actor to change into some other state at this point (perhaps after traversing a particular connector).

19.6.5. GuidedTourState

There may be occasions when you want an NPC to take the lead and have the player character invited to follow him or her. First, we'll give a brief description of how it works.

According to the comments in the library code:

Guided Tour state. This provides a simple way of defining a "guided tour," which is a series of locations to which we try to guide the player character. We don't force the player character to travel as specified; we merely try to lead the player. The actual travel is up to the player.

Here's how this works. For each location on the guided tour, create one of these state objects. Set `escortDest` to the travel connector to which we're attempting to guide the player character from the current location. Set `stateAfterEscort` to the state object for the next location on the tour. Set `stateDesc` to something indicating that we're trying to show the player to the next stop - something along the lines of "Bob waits for you by the door." Set `arrivingWithDesc` to a message indicating that we just showed up in the current location and are ready to show the player to the next - "Bob goes to the door and waits for you to follow him."

The significant new properties defined on `GuidedTourState` are:

- **`escortActor`** - the actor being escorted - this is usually the player character
- **`escortDest`** - the travel connector we're trying to show the player character into

TADS 3 Tour Guide

- **escortStateClass** - The class we use for our actor state during the escort travel. By default, we use the basic guided-tour accompanying travel state class ([GuidedInTravelState](#)), but games will probably want to use a customized subclass of this basic class in most cases. The main reason to use a custom subclass is to provide customized messages to describe the departure of the escorting actor.
- **stateAfterEscort** - The next state for our actor to assume after the travel. This should be overridden and set to the state object for the next stop on the tour.

Now we'll give a very simple example. Sarah has lost her ring, and if the player character asks her about it enough times, she'll lead him back inside the cave to look for it, decide she can't find it, and then lead him back out to the lakeside again. We can define a series of GuidedTourStates (which must obviously be located in Sarah), that define her leading the playing character on a brief and futile search for her lost ring:

```
+ sarahGuide1 : GuidedTourState
  stateAfterEscort = sarahGuide2
  escortDest = lakeDoor2
  stateDesc = "She's waiting for you by the door. "
  specialDesc { inherited; stateDesc; }
;

+ sarahGuide2 : GuidedTourState
  stateAfterEscort = sarahGuide3
  escortDest = mainCave
  arrivingWithDesc = "{The sarah/she} walks round the cave, scanning the
    floor and the furniture with a deep frown on her face, then shakes
    her head and walks back over to the door.<.p>
    <q>I can't see it here,</q> she says, <q>Perhaps it's in that
    big cave over there.</q> She starts walking north, then stops
    and turns, waiting for you to follow.<.p>"
  stateDesc = "She's standing looking at you, waiting for you
    to follow her into the large cave to the north. "
  specialDesc { inherited; stateDesc; }
;

+ sarahGuide3 : GuidedTourState
  stateAfterEscort = sarahGuide4
  escortDest = anotherCave
  arrivingWithDesc = "{The sarah/she} hunts round the cave, peering at
    the torch on the wall, the trunk on the floor, and the various
    ways out. Then she lets out a heavy sigh and declares, <q>Oh, it's
    <i>hopeless</i>! How can I find such a small thing here? Anyway,
    I'm sure he ran outside with it. Let's go back out by the lake.</q>\b
    So saying, she walks back towards the cave to the south, then stops,
    waiting for you to follow.<.p>"
  stateDesc = "She's waiting for you by the opening to the cave to
    the south. "
  specialDesc {inherited; stateDesc; }
;

+ sarahGuide4 : GuidedTourState
  stateAfterEscort = sarahTalking
  escortDest = lakeDoor
  arrivingWithDesc = "{The sarah/she} walks briskly over to the door,
    then stops to wait for you. "
  stateDesc = "She's waiting for you by the door. "
  specialDesc {inherited; stateDesc; }
;
```

There's one other refinement we can add her, and that's to add **TourGuide** to Sarah's class list, ahead of Person:

```
sarah : TourGuide, Person 'attractive young woman/brunette' 'young woman' @lakeRoom
...
;
```

This will allow the player to type FOLLOW SARAH or FOLLOW WOMAN and have her lead the way while she's in one of her GuidedTourStates.

There's only one problem: so far we've provided no means for her to *get* into these GuidedTourStates. Since in this case it will require a mechanism we haven't yet come to, we'll have to wait until we do, which will be when we start defining some [AskTopics](#) for Sarah.

19.6.6. GuidedInTravelState

The GuidedInTravelState is the version of AccompanyingInTravelState for use with the GuidedTourState. According to the comments in the library code, it is:

A subclass of the basic accompanying travel state specifically designed for guided tours. This is almost the same as the basic accompanying travel state, but provides customized messages to describe the departure of our associated actor, which is the actor serving as the tour guide.

This is how we might customize the messages from the last stage of Sarah's brief guided tour, making use of the GuidedInTravelState:

```
+ sarahGuide4 : GuidedTourState
  stateAfterEscort = sarahTalking
  escortDest = lakeDoor
  arrivingWithDesc = "{The sarah/she} walks briskly over to the door,
    then stops to wait for you. "
  stateDesc = "She's waiting for you by the door. "
  specialDesc {inherited; stateDesc; }
  escortStateClass = sarahGuidedState
;

class sarahGuidedState : GuidedInTravelState
  sayDepartingThroughPassage(conn)
  {
    "{The sarah/she} goes out through the door. ";
  }
  specialDesc = "{The sarah/she} stops by the lakeside and starts
    searching the ground once more. "
;

```

Note that we have to override sayDepartingThroughPassage(conn) here because Sarah will be going through the door.

19.6.7. InConversationState

The InConversationState is, as its name suggests, the state to use when an NPC is in conversation with the player character. Each InConversationState is normally associated with a [ConversationReadyState](#), and the library automatically handles switching between the two depending on whether a conversation is in progress or not. Addressing an actor who is in a ConversationReadyState will automatically cause the actor to switch into the related InConversationState. When a conversation is terminated, the actor switches back into the ConversationReadyState (or else some other state defined in the nextState property). The conversation is normally terminated for one of four reasons: (1) the PC explicitly says goodbye; (2) the PC moves to another location; (3) the PC fails to continue the conversation for a number of turns and the NPC's attention span is exhausted; (4) the NPC decides to terminate the conversation.

In addition to the methods and properties defined on ActorState, which InConversationState inherits, the properties and methods you are most likely to want to work with are:

- **attentionSpan** - this is an integer giving the number of turns the actor should wait before giving up on the conversation. The default is 4. If the other character doesn't talk to the NPC for this many turns, the NPC will automatically terminate the conversation, switching to the next state. If you want the NPC's attention span to be infinite, set attentionSpan to nil.
- **nextState** - this is an ActorState object, which should normally be of the ConversationReadyState class, which follows the conversation's termination. When we terminate the conversation, we'll switch to this state. You don't have to override this; if you don't, the game will remember the state that the actor was in just before the conversation, and switch back to that when the conversation ends.
- **endConversation (actor, reason)** - Exit the in-conversation state. 'reason' indicates why we're leaving the conversation - this is one of the [endConvXxx](#) enums defined in adv3.h. This method is a convenience only; you aren't required to call this method to end the conversation, since you can simply switch to another actor state directly if you prefer. This method's main purpose is to display an appropriate message terminating the conversation while switching to the new state. If you want to display your own message directly from the code that's changing the state, there's no reason to call this. This returns true if you want to allow the conversation to end, nil if not.

TADS 3 Tour Guide

Armed with information, we can set about creating `InConversationState` objects for various of our NPCs, starting with Sarah (so this should be nested in the `sarah` object):

```
+ sarahTalking : InConversationState
  specialDesc = "Sarah is looking at you, waiting for you to speak. "
  stateDesc = "She's waiting for you to speak. "
  attentionSpan = 5
;
```

Similarly we can define for the curator (remembering to make sure it's nested in the `curator` object):

```
+ curatorTalking : InConversationState
  stateDesc = "He's looking straight at you, head cocked slightly to one side. "
  specialDesc { inherited; stateDesc; }
  attentionSpan = 3
;
```

And likewise for King Solomon, who, being so famous for his wisdom, and not being caught up in the frenetic pace of modern living, we'll give a rather greater attention span to:

```
+ solomonTalking : InConversationState
  specialDesc = "{The solomon/he} is looking up at you with a thoughtful expression
  on his face. "
  stateDesc = "He's looking up at you. "
  attentionSpan = 20
;
```

Conversely, we'll assume that the demons are peculiarly impatient, and also not at all keen on letting you progress down to the plain:

```
+ demonsChattering : InConversationState
  specialDesc = "The demons are chattering away madly as they block your path. "
  stateDesc = "They're chattering and blocking your path at the same time. "
  attentionSpan = 1
  beforeTravel(traveler, connector)
  {
    if(connector == hellPathDown)
    {
      "The demons crowd in on you, chattering away and blocking your path
      all the more insistently. <q>You can't come this way! You can't come
      this way!</q> they chorus. <q>We'll eat your flesh! We'll eat your flesh!</q>
      some of them add, leering at you hungrily. ";
      exit;
    }
    inherited(traveler, connector);
  }
;
```

Obviously, this code needs to be placed just after the `demons` object.

Finally, we'll give our ghost an `InConversationState`:

```
+ ghostTalking : InConversationState
  specialDesc = "The ghost is hovering over its tomb, staring at you intently. "
  stateDesc = "It's hovering over its tomb. "
;
```

19.6.8. ConversationReadyState

A `ConversationReadyState`, as its name suggests, is the kind of `ActorState` is in when the actor is not actually talking to the PC but is prepared to do so. When a conversational command (ASK ABOUT, TELL ABOUT, ASK FOR, SHOW TO, GIVE TO, TALK TO) is addressed to an actor who's in a `ConversationReadyState`, the actor automatically switches into the associated `InConversationState`. What the associated state is may be defined either explicitly by setting the **inConvState** property, or implicitly by nesting the `ConversationReadyState` inside its associated

TADS 3 Tour Guide

InConversationState. Here we'll use the latter method, even though here, as typically, the ConversationReadyStates we'll be defining are the initial states of the actors in question.

We'll continue where we left off, with the demons (who must still block your path in this state too). This code follows immediately after the demonsChattering state:

```
++ demonsPrancing : ConversationReadyState
specialDesc = "A gaggle of demons is prancing around on the rocks near the steps down
towards the basalt plain, pretending to ignore you, but keeping a careful watch on
you just the same. "
stateDesc = "They're prancing around near the steps down to the plain. "
isInitState = true
beforeTravel(traveler, connector)
{
    if(connector == hellPathDown)
    {
        "The demons bunch together and shriek at you, driving you back from the steps. ";
        exit;
    }
    inherited(traveler, connector);
}
;
```

The initial ConversationReady states for Sarah and Solomon are rather more straightforward (again, each one should be nested within the corresponding InConversationState):

```
++ sarahLooking : ConversationReadyState
isInitState = true
specialDesc = "{A sarah/she} is standing by the shore, apparently
looking at something, though she occasionally throws a curious
glance your way. "
stateDesc = "She seems to be looking for something. "
;

++ solomonExamining : ConversationReadyState
specialDesc {inherited; stateDesc; }
stateDesc = "He's staring at the table, deep in thought. "
isInitState = true
;
```

While that of the curator is a little more complicated:

```
++ curatorWatching : ConversationReadyState
stateDesc = "He's watching you carefully. "
specialDesc { inherited; stateDesc; }
isInitState = true
beforeTravel(traveler, connector)
{
    inherited(traveler, connector);
    if(traveler == gPlayerChar)
    {
        switch(connector)
        {
            case museum:
                getActor.moveIntoForTravel(byCases);
                "{The curator/he} follows you into the museum. ";
                break;
            case museumLobby:
                getActor.moveIntoForTravel(behindTable);
                "{The curator/he} follows you into the lobby. ";
                break;
            case benefactorsRoom:
                "<q>I'm afraid you can't go in there,</q> {the curator/he}
intercepts you, <q>That's the <i>benefactors'</i> room; only
our benefactors are allowed in there to see the special
exhibits.</q> ";
                exit;
        }
    }
}
;
```


TADS 3 Tour Guide

In order to make this travel checking occur on the related `InConversationState` without having to repeat all the code we can simply call this method thus:

```
+ curatorTalking : InConversationState
  stateDesc = "He's looking straight at you, head cocked slightly to one side. "
  specialDesc { inherited; stateDesc; }
  attentionSpan = 3
  beforeTravel(traveler, connector)
  {
    curatorWatching.beforeTravel(traveler, connector);
    inherited(traveler, connector);
  }
;
```

19.6.9. Greeting Protocols

To walk up to an NPC and start a conversation with an ASK ABOUT, and then, after a few more conversational commands, to terminate the conversation by simply walking away or ceasing to converse can make the game transcript seem a bit abrupt and not very life-like. In real life one normally starts a conversation with "Hello" or "Excuse me" or something else designed to open the channel of communication, and then concludes it with a similar winding-up exchange to signal that it's now over.

In order to handle the beginning and end of conversations, you put a [HelloTopic](#) and a [ByeTopic](#) in the [ConversationReadyState](#), for example:

```
++ sarahLooking : ConversationReadyState
  isInitState = true
  specialDesc = "{A sarah/she} is standing by the shore, apparently
    looking at something, though she occasionally throws a curious
    glance your way. "
  stateDesc = "She seems to be looking for something. "
;

+++ HelloTopic, ShuffledEventList
[
  '<q>Hello there,</q> you say.\b
  <q>Hello.</q> she smiles at you, slightly quizzically.'
]
[
  '<q>Hello again.</q> you greet her.\b
  <q>Hi!</q> she replies. ',
  '<q>It\'s me again.</q> you tell her.\b
  <q>So I noticed.</q> she answers.',
  '<q>Er...</q> you start.\b
  <q>Yes?</q> she asks. '
]
;

+++ ByeTopic
  "<q>Cheerio, then!</q> you say.\b
  <q>See you!</q> she replies. "
;

+++ ImpByeTopic
  "{The sarah/she} gives a little shrug and goes back to
  searching her surroundings. "
;
```

The effect of this is that when the player character strikes up a conversation with Sarah, either explicitly through a GREET SARAH, TALK TO SARAH or SARAH, HELLO command, or implicitly by addressing a conversational command to her like ASK SARAH ABOUT HERSELF or TELL SARAH ABOUT BANANA, the greeting message from the [HelloTopic](#) will be displayed, and then Sarah will switch into the associated `InConversationState`. Conversely, once the conversation is over, Sarah will switch back into the `ConversationReadyState` and either the [ByeTopic](#) or the [ImpByeTopic](#) will be displayed, depending on whether the player ended the conversation explicitly with a BYE command, or implicitly by either leaving the location or failing to address a conversational command to Sarah for the number of turns needed to exhaust her attention span.

We'll be giving more examples of this later.

19.7. Topic Entries

19.7.1. TopicEntry

One way to handle conversational commands such as ASK FRED ABOUT TREASURE or SHOW ROTTEN APPLE TO MAVIS might be to override the `actionlobjAskAbout` and `actionlobjShowTo` methods on Fred and Mavis with long and complicated statements, probably starting with `switch(gDobj)` and then going on to have complex IF ... ELSE IF spaghetti within the more complex cases, until the whole thing becomes a nightmare to code, an impossibility to debug, and a disincentive to attempt any kind of sophisticated conversation with an NPC. Fortunately TADS 3 makes such spaghetti nightmares a thing of the past, thanks mainly to the `TopicEntry`.

To handle conversations in TADS 3 there is no need to override a single action method on any NPC (unless you're attempting something not covered by the library). The commands ASK ABOUT, ASK FOR, TELL ABOUT, SHOW TO and GIVE TO can all be handled with `TopicEntry` objects (as can some other constructs we'll be meeting later). These objects define the response an actor gives to a particular command concerning a particular object (for example an object defined as `AskTopic @carbuncle` would define a response to an ASK ABOUT CARBUNCLE command). `TopicEntry` objects may be nested directly inside the Actor whose responses they represent, but they may also be nested inside `ActorStates`, to form the set of responses the Actor gives when in that state, or else within `TopicGroups`, to form a set of responses the actor gives when certain conditions obtain. `TopicEntry` objects may also be placed within `TopicGroups` within `ActorStates`. Again, each `TopicEntry` may be allocated a condition that must obtain before it is used as a response, or may be assigned one or more alternative entries (`AltTopics`) that will be used under different conditions. Again, a `TopicEntry` may (depending on circumstances) match a physical game object (such as the carbuncle), an abstract topic (such as 'the meaning of life'), a set of objects or topics, or a regular expression. If none of these is suitable, the author can override the `TopicEntry`'s **`matchTopic`** method to provide custom matching (for example, to match any object belonging to a certain class). As if that were not enough, the `TopicEntry` system can also cater for NPCs being proactive and posing questions to the player character, to which any set of responses may be defined (via `SpecialTopic`).

The above explanation is probably too abstract and dense to be terribly meaningful at first reading. For a more digestible explanation you may want to read the article on 'Programming Conversations with NPCs' in the *Technical Manual*. In any case, hopefully the dense, abstract explanation given above will begin to make more sense when we start giving concrete examples.

The first thing to note is that game authors are most unlikely to use raw `TopicEntry` objects (unless they're devising a custom `TopicEntry` class of their own). In practice one uses one of the subclasses of `TopicEntry`:

```
TopicEntry
  AltTopic
  CommandTopic
  DefaultTopic
    DefaultAskForTopic
    DefaultAskTellTopic
    DefaultAskTopic
    DefaultCommandTopic
    DefaultConsultTopic
    DefaultGiveShowTopic
    DefaultGiveTopic
    DefaultInitiateTopic
    DefaultShowTopic
    DefaultTellTopic
  SpecialTopic
  ThingMatchTopic
    GiveShowTopic
      GiveTopic
      ShowTopic
    InitiateTopic
  TopicMatchTopic
    AskTellTopic
    AskForAboutTopic
```

TADS 3 Tour Guide

```
AskForTopic
AskTellAboutForTopic
AskTopic
TellTopic
ConsultTopic
TopicOrThingMatchTopic
AskTellGiveShowTopic
AskTellShowTopic
MiscTopic
ByeTopic
HelloGoodbyeTopic
HelloTopic
ImpByeTopic
ImpHelloTopic
YesNoTopic
NoTopic
YesTopic
```

The structure of this inheritance tree is important, because it gives a clear indication of the different subcategories of TopicEntry, each one of which is subtly different and has a slightly different use and purpose. The first distinction to be clear about is between ThingMatchTopics and TopicMatchTopics; the former match game objects (generally of class Thing or one of its subclasses) while the latter match ResolvedTopics. The main consequence of this is that a ThingMatchTopic can only match an object that's in scope (this makes sense - you can't give or show an object to someone unless the object is there to be given or shown), whereas a TopicMatchTopic can match any object that's been defined (including a Topic), since something doesn't have to be physically present to be asked about, asked for, or told about. The distinction between ThingMatchTopic and TopicMatchTopic is also important if you override certain of their methods, as we'll see presently.

We'll come across the other main types of TopicEntry in due course. [AltTopic](#) is used to provide an alternative response to another TopicEntry under author-defined conditions. [DefaultTopics](#) match any input relating to the command in question, (i.e. a DefaultAskTopic matches ASK FRED ABOUT X whatever X is), and is activated by anything for which a more specific response has not been defined. [SpecialTopic](#) responds to any author-defined string (e.g. PRAISE SARAH, TELL CURATOR A LIE or RECITE A POEM), but works only in the context of a Conversation Node.

TopicEntries have the following methods and properties which you either will or may need to use:

- **matchObj** - The matching simulation object or objects; this can either be a single object or a list of objects.
- **matchPattern** - a regular expression pattern that this TopicEntry matches (as an alternative to matchObj)
- **matchScore** - the match strength score. By default this is 100 (except on DefaultTopics, where it's 1, 2 or 3). If more than one TopicEntry is a possible match to a conversational command, the one with the highest matchScore is used.
- **getActor()** - The Actor object to which this TopicEntry ultimately belongs (you'll never want to override this but you may want to refer to it).
- **topicResponse** - The response to ASK, TELL, GIVE or SHOW. You override this to show the response text (normally in a double-quoted string, but you can define this as a method if you need something more complicated), alternatively you can override handleTopic to do the job.
- **isActive** - The condition that must be true for this TopicEntry to be matched. This is true by default but can be set to anything you like to make matching conditional; while this property evaluates to nil its response will not be shown. For example, you might want a particular response to be used only if Sarah has seen the ghost, in which case you could define isActive = sarah.hasSeen(ghost).
- **isConversational** - true by default, this property determines whether this response is treated as conversational. If this property is set to nil then matching this topic will not trigger any greeting protocols. For example, if the player attempts to ASK SARAH FOR BANANA when the PC is already carrying the banana, instead of having Sarah deliver a sarcastic reply you may simply want the game to report "You already have the banana." Since this would not constitute a conversational exchange between Sarah and the PC you'd probably want to set isConversational = nil on this response.
- **handleTopic(fromActor, topic)** - By default this either calls doScript (if the TopicEntry also inherits from a Script class - e.g. if it was defined as AskTopic, StopEventList) or else (if the TopicEntry is not also a Script) simply calls topicResponse. The first parameter (fromActor) is the actor (normally the player character) doing the asking, showing, giving or telling. The meaning of the second parameter depends on the type of TopicEntry. For a ThingMatchTopic (GiveTopic, ShowTopic or InitiateTopic) the parameter is the actual game object that the TopicEntry matched (this can be useful, e.g. in a Give command if you want to move the object matched into the actor it's been given to). For a TopicMatchTopic (AskTopic, TellTopic, AskForTopic), however, the second parameter is a ResolvedTopic object (though you may be able to get at the actual game object matched by calling topic.getBestMatch).

TADS 3 Tour Guide

- **matchTopic(fromActor, topic)** - By default this matches the TopicEntry according to what you have defined in matchObj or matchPattern, so you don't need to worry about overriding it. On occasion, however, you may have a special case that's most easily dealt with by overriding this method. For example, if you defined a custom Coin class and then wanted a GiveTopic that matched any member of the Coin class you could override matchTopic(fromActor, obj) to return matchScore if obj is ofKind(Coin). Note that the parameters to this method mean the same as those for handleTopic, so that the meaning of the second parameter depends on whether we're using a TopicMatchTopic or a ThingMatchTopic.
- **isMatchPossible(actor, scopeList)** - This method decides whether the TopicEntry can be matched. For a TopicMatchTopic it returns true if the matchObj (or any of the objects in the matchObj list if it is a list) is either in scope (in the scopeList) or is known to actor, or if matchObj is nil (which means that the TopicEntry is being matched by a pattern or a custom method). For a ThingMatchTopic it returns true only if matchObj or one of the objects in the matchObj list is in scope. Most of the time you probably won't want to change this behaviour.

This may look a bit daunting, but in practice the use of templates makes the definition of the majority of TopicEntries pretty straightforward. To ask about a single object, for example, one might define an AskTopic thus:

```
+ AskTopic @cheese
  "<q>Do you like cheese?</q> you ask.\b
  <q>Only if it's blue cheese.</q> she replies. "
;
```

The object following the @ sign is the matchObj, and the double-quoted string that follows is the topicResponse.

If you want a TopicEntry that matches a list of items, you enclose that list in square brackets instead of using the @ sign, e.g.

```
+ AskTopic [cheese, bread]
  "<q>Do you like cheese on your bread?</q> you enquire.\b
  <q>That depends on the cheese.</q> she tells you."
;
```

If you wanted a different response that takes preference under special circumstances you could use a higher matchScore by placing it after a + sign before the matchObj:

```
+ AskTopic +110 @cheese
  "<q>Do you like this cheese?</q> you ask, holding out the lump you're holding.\b
  <q>Ooh! Blue Stilton! lovely!</q> she exclaims. "
  isActive = (cheese.isHeldBy(gPlayerChar))
;
```

In all these examples the player will always see the same response to the same command. For more realism you may prefer to use a list of responses, which you can do by adding a Script class to the class list of the TopicEntry and supplying a list of responses in square brackets instead of a single response in a double-quoted string:

```
+ AskTopic, StopEventList [cheese, bread]
[ '<q>Do you like cheese on your bread?</q> you ask.\b
  <q>That depends on the cheese</q> she tells you. ',

  '<q>So, what kind of cheese do you like with your bread?</q> you enquire.\b
  <q>Blue cheese.</q> she replies firmly. ',

  '<q>I can't interest you in some nice gooey brie on a fresh crusty piece
  of wholemeal then?</q> you suggest\b
  <q>No - but if you made that a nice smelly piece of Blue Stilton on a
  fresh crusty piece of wholemeal that'd be a different matter.</q> she smiles. ',

  '<q>So - only blue cheese on your bread will do.</q> you surmise.\b
  <q>That\'s right.</q> she affirms. '
]
```

Finally, you might want to override topicResponse with a method if it needs to do something more complicated than merely displaying a piece of text, for example:

```
+ GiveTopic @cheese
  topicResponse
  {
    "<q>Here, have this,</q> you say, handing over the lump of stilton.\b
```

TADS 3 Tour Guide

```
<q>Thanks!</q> she beams, then proceeds to eat the lot. "
cheese.moveInto(getActor);
nestedActorAction(getActor, Eat, cheese);
}
;
```

Having looked at TopicEntries in the abstract, we can now go on to see how we might use the various types of TopicEntry in our game.

19.7.2. GiveTopic

We use a GiveTopic to handle GIVE TO commands, instead of writing iobjFor(GiveTo) handlers on the actor. Like other TopicEntries, GiveTopics can be put in particular ActorStates so that they represent the handling particular to that state.

We can now rip out all the iobjFor(GiveTo) handling from the sarah object, and achieve the same result with a GiveTopic instead:

```
sarah : Person 'attractive young woman/brunette' 'young woman' @lakeRoom
  "She's an attractive brunette, somewhere in her mid-twenties. "
  isHer = true
  properName = 'Sarah'
  globalParamName = 'sarah'
;
...

+ sarahTalking : InConversationState
  specialDesc = "Sarah is looking at you, waiting for you to speak. "
  stateDesc = "She's waiting for you to speak. "
  attentionSpan = 5
;

++ GiveTopic @diamondRing
  topicResponse
  {
    "<q>Great! Thanks!</q> {the sarah/she} declares, <q>I've been looking for it
    <i>every</i>where!</q> She slips the ring on her finger, then continues,
    <q>Well, I suppose the next thing is to find a way out of here. You seem to
    know how to find things, so I guess I'd best follow you.</q><.p>";
    gDobj.moveInto(getActor);
    gDobj.makeWornBy(getActor);
    getActor.setCurState(sarahFollowing);
  }
;
```

At some point the player will need to give Sarah one of the gas masks if she's to follow him into Hell Fire Cavern. The problem here is that we have two identical gas masks and we could equally well give her either of them. Again, we don't know what ActorState she'll be in when we give the gas mask (it's likely to be the sarahFollowing state, but the player character could conceivably hand Sarah a gas mask before giving her the diamond ring that makes her start following him around). For this reason it might be safer to put this GiveTopic directly inside the sarah object (where it'll be accessible from all of her ActorStates). Be careful with the nesting though - don't put it between an ActorState and something that belongs inside that ActorState, or things will start to go awry. To solve the two gas-mask problem we need to write custom matchTopic and handleTopic methods (note that the kind of method we've used here will only work on TopicEntries that handle resolved objects rather than resolved topics, that is GiveTopic, ShowTopic, GiveShowTopic and their Default equivalents):

```
+ GiveTopic
  matchTopic(fromActor, obj)
  { return obj.ofKind(GasMask) ? matchScore : nil; }
  handleTopic(fromActor, obj)
  {
    obj.moveInto(getActor);
    "<q>Thanks,</q> {the sarah/she} remarks dubiously as she accepts it from
    you, <q>I'm sure it'll - er - come in very useful.</q> ";
  }
;
```

TADS 3 Tour Guide

The curator is anxious to add Solomon's purple carbuncle to his collection, so we can add an appropriate GiveTopic to the curator's curatorTalking state:

```
++ GiveTopic @carbuncle
topicResponse
{
    "{The curator/he} takes the carbuncle and examines it carefully, then declares,
    <q>Wunderbar! Ausgezeichnet! This is the famous purple carbuncle of King Solomon,
    nicht wahr? And you are giving it to the museum? How kind, how very kind!</q>
    Pausing just to wipe the tears of excitement and gratitude out of his eyes, he
    continues, <q>I shall enroll you on our roll of honoured benefactors <i>at once</i>!
    Please, please, do feel free to inspect the special treasures in our benefactors'
    exhibition room any time you please!</q>";
    carbuncle.moveInto(getActor);
}
;
```

Of course giving the wrong thing to the wrong people could have disastrous consequences. If the player is foolish enough to hand the banana over to the demons, he or she richly deserves to lose, so we can code this in a GiveTopic (which, like all the TopicEntries relating to the demons, should be placed under the demonsChattering state).

```
++ GiveTopic @goldenBanana
topicResponse
{
    "<q>Yes!</q> the demons cry as you hand them the Golden Banana of Discord.
    <q>It's back where it belongs - with us, the demonic cabal!</q>\b
    It dawns on you that you may just have made a ghastly mistake - probably
    the worst you've ever made.\b
    But at least it'll probably be your last.\b ";
    endGame('YOU HAVE FAILED DISMALLY');
}
;
```

19.7.3. ShowTopic

A ShowTopic is very similar to a GiveTopic, except that it handles the SHOW TO command rather than the GIVE TO command. We might, for example, use it to display a different response if the player character merely shows Sarah her lost diamond ring rather than returning it to her:

```
++ ShowTopic @diamondRing
"{The sarah/she} inspects the ring then looks up at you, <q>Yes, that's my ring!</q>
she declares, <q>May I have it, please?</q>"
;
```

Note that in this case, the topicResponse is simply a double-quoted string, which the TopicEntry template can cope with, so we don't need to write out topicEntry as an explicit method. However, in this case the response is a bit too simplistic, since Sarah will keep repeating the response each time she's shown the ring. It would be better if we used, say, a StopEventList to show a sequence of responses if the player character keeps showing Sarah the ring without giving it to her:

```
++ ShowTopic, StopEventList @diamondRing
[
    '{The sarah/she} inspects the ring then looks up at you, <q>Yes, that\'s my ring!</q>
    she declares, <q>May I have it, please?</q>',
    '<q>I think we\'ve already established that\'s my ring.</q> she points out, <q>I\'d
    like it back now, please.</q>',
    '{The sarah/she} holds out her hand to you with a look of one doing her best to
    bear patiently with someone unusually slow of understanding. '
]
;
```

On the other hand, perhaps we can get away with one response should the player show the carbuncle to the curator:

```
++ ShowTopic @carbuncle
```

TADS 3 Tour Guide

```
"{The curator/he} stares at the carbuncle in your hand, <q>Is that what I think  
it is?</q> he wonders, <q>May I have a closer look?</q>"
```

You may recall a little way back we left the path down into Hellfire Cavern blocked by a nasty bunch of demons. The Baaras root Solomon is busily studying is meant to be good for disposing of demons (or so Josephus tells us), so perhaps showing it to the demons will do the trick:

```
++ GiveShowTopic @baarasRoot  
  topicResponse  
  {  
    "As you produce the baaras root and hold it up before their demonic little  
    eyes, it starts to glow an eerie pink colour. <q>Begone foul fiends!</q> you  
    cry, <q>By the power of Solomon's Baaras Root, and with the incantation he  
    forgot to teach me - er - I banish all evil spirits, all demons and creatures unclean,  
    into the Almighty Garbage Collection routine whence no Dangling Reference shall  
    ever return!</q>\b  
    <q>Ah no! Mandragora Maxima!</q> cries one of the demons.\n  
    <q>Eek no! The forgotten incantation!</q> shrieks another.\n  
    <q>Fie and double discombobulation, the Almighty Garbage Collector!</q> squeals  
    another.\n  
    <q>RunDaemon, RunDaemon!</q> yells a fourth.\b  
    As the demons turn in terror and try to flee, the Baaras root  
    grows ever brighter in your hand, shedding its piercing pink rays over the  
    demonic horde, so that even as they start to clamber down the slope they  
    shimmer and dissolve, turning into plumes of oily black smoke which  
    vanishes like a mist. ";  
    demons.moveInto(nil);  
  }  
;
```

On the other hand, showing them the banana will get them worked up for other reasons:

```
++ ShowTopic @goldenBanana  
  "As you hold up the Golden Banana of Discord for the demons to see they  
  become very excited indeed. <q>Give us our Banana back!</q> they cry,  
  <q>Yes, we have no banana, we want our banana today!</q> they sing,  
  <q>Gimme! Gimme! Gimme!</q> they insist.\b  
  <q>Pretty please?</q> one of them adds plaintively. "  
;
```

Since showing the Baaras root to the demons clears them out of the way, we ought to code the locations the player character will then be able to reach:

```
basaltPlain : OutdoorRoom 'Basalt Plain' 'the basalt plain'  
  "To the north this rough, grey basalt plain is bordered by the fiery volcano;  
  to the south it comes to the end at a steep rocky slope, which can be ascended  
  by means of steep stone steps. Progress across the bleak plain looks difficult,  
  since the ground is broken, pitted and uneven. "  
  south asExit(up)  
  north = baseOfVolcano  
  east : FakeConnector { "You struggle a few dozen yards to the east, but the effort  
    seems so unrewarding that you quickly turn back. " }  
  west : FakeConnector { "You stumble over the broken ground to the west, but there  
    doesn't appear to be anything interesting in that direction so you quickly  
    abandon the attempt as futile. " }  
  up = basaltPathUp  
;  
  
+ basaltPathUp : StairwayUp -> hellPathDown 'stone upward up steps' 'steep stone steps'  
  isPlural = true  
;  
  
baseOfVolcano : OutdoorRoom 'Base of Volcano' 'the base of the volcano'  
  "The harsh basalt plain to the south comes to the end at the base of a  
  volcanic mountain that's busily belching flames and smoke. A narrow path  
  leads uninvitingly up the side of the volcano. "  
  south = basaltPlain  
  north asExit(up)  
  up = volcanoPath  
;
```

TADS 3 Tour Guide

```
+ volcanoPath : StairwayUp ->volcanoPathDown 'narrow up path' 'narrow path up'
;
```

At this point you might want to add these two new rooms to the Distant volcano object's locationList:

```
MultiLoc, Distant 'mount volcano/gloom' 'volcano'
    "The volcano rises up from the basalt plain like an angry sore, belching fumes,
    smoke and occasional balls of lava, which spit from the summit and ooze
    pus-like down its rugged slopes. "
    locationList = [hellFireCavern, hellPath, basaltPlain, baseOfVolcano]
;
```

Note at this point we have finally connected up the full path to the lava pool where the Golden Banana of Discord is to be destroyed.

19.7.4. GiveShowTopic

In many, perhaps most situations, we may not really want to distinguish between GIVE x TO y and SHOW x TO y. In such situations, where we want both commands to be treated exactly the same, we can use a GiveShowTopic. For example, if the player tries to return either the ring or the diamond to Sarah before attaching them together to restore the diamond ring, we might want Sarah to refuse politely and ask the player to complete the task:

```
++ GiveShowTopic @diamond
    "{The sarah/she} studies the gem carefully, <q>That certainly looks like it could
    be the diamond from my ring,</q> she decides, <q>But where's the ring?</q>"
;

++ GiveShowTopic @ring
    "{The sarah/she} nods eagerly, <q>Yes, that's my ring!</q> she declares, but then
    her hand flies to her mouth, <q>But - oh my goodness - the diamond is missing!</q>"
;
```

If you try this out, however, it will rapidly become obvious that it generates something of an unrealistic succession if Sarah is shown both the ring (sans diamond) and the diamond (sans ring) in either order. We'll see how to fix that shortly using [AltTopic](#). In the meantime let's add some more GiveShowTopics.

You'll remember some time ago we installed a vending machine to dispense museum tickets. Our curator really ought to check whether the player character has a ticket before allowing him into the museum. It would be tedious to enforce this check every time the player character enters the museum, however, so we'll make it a once and for all check (which is reasonable; the ticket allows multiple entries and the curator can remember being shown it). Again, players shouldn't guess whether they have to GIVE the ticket or SHOW the ticket, so it's a good candidate for a GiveShowTopic. We could make the GiveShowTopic set a custom property of the curator to signal that the ticket has been shown, but it's actually much simpler to use the `<.reveal>` tag to set a named key that does the same job. Our GiveShowTopic (belonging to the curator's curatorTalking state) then looks like this:

```
++ GiveShowTopic @museumTicket
    "<q>Thank you, that's fine.</q> {the curator/he} nods as he inspects your ticket,
    <q>Enjoy the exhibits!</q><.reveal ticket-shown>"
;
```

To test that the 'ticket-shown' flag has been set, we use `gRevealed('ticket-shown')`. We need to amend the code in `curatorWatching` to make this new check, and while we're at it, we'll change it so the curator will let the player character into the benefactors' room if he (the curator) has been given the carbuncle (which makes the player character a benefactor of the museum), except that the curator won't let anyone into the benefactors' room with any kind of container in which the golden banana might be smuggled out (apart from the cap, which the curator fails to recognize as a potential container, or the Matchbook, which no one would regard as a container, though it does inherit from the Container class).

```
++ curatorWatching : ConversationReadyState
    stateDesc = "He's watching you carefully. "
    specialDesc { inherited; stateDesc; }
    isInitState = true
```


TADS 3 Tour Guide

```
beforeTravel(traveler, connector)
{
    inherited(traveler, connector);
    if(traveler == gPlayerChar)
    {
        switch(connector)
        {
            case museum:
                if(gRevealed('ticket-shown'))
                {
                    getActor.moveIntoForTravel(byCases);
                    "{The curator/he} follows you into the museum. ";
                }
                else
                {
                    "{The curator/he} stops you, asking, <q>May I see your
                    ticket please?</q><.p>";
                    exit;
                }
                break;
            case museumLobby:
                getActor.moveIntoForTravel(behindTable);
                "{The curator/he} follows you into the lobby. ";
                break;
            case benefactorsRoom:
                if(!carbuncle.isIn(getActor))
                {
                    "<q>I'm afraid you can't go in there,</q> {the curator/he}
                    intercepts you, <q>That's the <i>benefactors'</i> room; only
                    our benefactors are allowed in there to see the special
                    exhibits.</q> ";
                    exit;
                }
                foreach(local cur in traveler.contents)
                if(cur.ofKind(Container) && cur != cap && !cur.ofKind(Matchbook))
                {
                    "<q>Sorry,</q> {the curator/he} apologizes as he intercepts
                    you, <q>But we can't allow you to take any bags or containers
                    in there. It's policy, I'm afraid - one can't be too careful
                    these days.</q> ";
                    exit;
                }
            }
        }
    }
}
```

You may think that this `beforeTravel` method is beginning to get bloated with spaghetti - or at least with switch statements and if statements, and perhaps you feel it's a shame that TADS 3 couldn't find some way to avoid this sort of thing in this case too. Well, if you are thinking that, you'll be glad to know that TADS 3 does provide an alternative here, which we'll look at shortly when we come to discuss [InitiateTopic](#).

King Solomon, meanwhile, is fairly anxious to recover a certain bronze bowl he's mislaid, so we need to provide for handing it over to him. The following `GiveShowTopic` belongs in the `solomonTalking` state:

```
++ GiveShowTopic @bronzeBowl
topicResponse
{
    "{The solomon/he}'s eyes light up as he spies the bronze bowl. <q>Excellent!</q>
    he declares, <q>Now I can look old Hiram in the eye when he comes! Well, I believe
    the traditional reward for a delighted monarch to give for service well done is
    <q>up to half my kingdom</q>, but being a wiser monarch than most I can see that's
    clearly far too extravagant. Still, perhaps some there's some more modest reward
    that would interest you?</q>\b
    {The solomon/he} examines the bowl and then carefully puts it down on the table.
    <.reveal bowl-returned> ";
    bronzeBowl.moveInto(solomonTable);
}
```

The player may also try to show or give this bowl to the curator, who, however, will prove uninterested; this belongs in `curatorTalking`:

TADS 3 Tour Guide

```
++ GiveShowTopic @bronzeBowl
  "{The curator/he} takes the bowl, turns it over, and gives it a cursory examination.
  <q>It's obviously very old,</q> he announces, <q>probably of middle eastern
  provenance. Possibly a valuable item, but not the sort of thing we're interested
  in here.</q> So saying, he hands it back. "
;
```

In the course of discovering the bronze bowl the player will find a number of gold coins which he may also want to show to the curator. This is more complex since we created three gold coins as anonymous objects of the GoldCoin class. Here we can use essentially the same method we employed with the gas masks by overriding matchTopic to test for the class of object being shown or given:

```
++ GiveShowTopic, StopEventList
[
  '{The curator/he} takes a brief look at the coin then hands it back. <q>It\'s
  just an old Roman coin,</q> he remarks dismissively, <q>We don\'t do coins
  here.</q>',
  '<q>I told you, this museum is not interested in old coins.</q> he reminds you.'
]
matchTopic(fromActor, obj)
{ return obj.ofKind(GoldCoin) ? matchScore : nil; }
;
```

We could use a similar technique for the tablets, which are sufficiently noteworthy that the curator might be expected to show some kind of interest in them. In this case we'll keep a note of which of the tablets the curator has already seen so that he makes an appropriate response depending on whether he has seen none of the tablets before, other tablets but not the one we're showing him, or the same one again:

```
++ GiveShowTopic
  handleTopic(fromActor, obj)
  {
    /* note the invocation */
    noteInvocation(fromActor);

    /* set pronoun antecedents if possible */
    setTopicPronouns(fromActor, obj);

    if(tabletsSeen.indexOf(obj))
      "<q>You've already shown me this <<obj.name>>.</q> he points out. ";
    else
    {
      if(tabletsSeen.length==0)
      {
        "{The curator/he} takes {the dobj/him} and examines it carefully. <q>Very interesting,</q>
        he remarks as he hands it back to you, <q>I don\'t exactly know what it is, but it\'s
        definitely interesting. Unfortunately, without knowing what it is, I can't give it
        a place in the museum.</q>";
      }
      else
      {
        "{The curator/he} examines {the dobj/him} with evident curiosity. <q>Another one!</q>
        he declares, <q>How very curious!</q>";
      }
      tabletsSeen += obj;
    }
  }
  matchTopic(fromActor, obj)
  { return obj.ofKind(Tablet) ? matchScore : nil; }
  tabletsSeen = []
;
```

TADS 3 Tour Guide

19.7.5. AltTopic

We use AltTopic topic to provide one or more alternative responses to conversational commands (ASK ABOUT, ASK FOR, TELL ABOUT, SHOW TO, GIVE TO) depending on some condition specified in the AltTopic's isActive property. AltTopics are nested within their corresponding TopicEntry, and the order in which they appear is significant: the last entry for which isActive is true is the one that is used. For example, supposed one had this structure:

```
bob : Person 'bob' 'Bob'
    ...
;

+ bobTalking : InConversationState
    ...
;

++ ShowTopic @letter
    "Bob glances at the envelope and then pushes it away, <q>I don't read other people's
correspondence</q>
    he says."
;

+++ AltTopic
    "Bob snatches the letter from your grasp and peruses it eagerly before handing it back.
<q>My goodness!</q> he declares.
    isActive = (gRevealed('exciting-scandal'))
;

+++ AltTopic
    "Bob stares at the letter disdainfully. <q>That's not much use now...</q> he complains."
    isActive = (letter.isTorn)
;
```

If we show the letter to bob when letter.isTorn is nil and the exciting scandal is yet to be revealed, then we'll see the "Bob glances at the envelope and pushes it away" response. If somewhere in the game <.reveal exciting-scandal> or gReveal('exciting-scandal') has been used, but letter.isTorn is still nil, then we'll see the "Bob snatches the letter from your grasp" response. If, however, letter.isTorn is true, we'll see the "Bob stares at the letter disdainfully" response regardless of the value of gRevealed('exciting-scandal').

Let's now put this to good use with Sarah, the ring, and the diamond:

```
++ GiveShowTopic @diamond
    "{The sarah/she} studies the gem carefully, <q>That certainly looks like it could
be the diamond from my ring,</q> she decides, <q>But where's the ring?</q>
    <.reveal diamond-shown>"
;

+++ AltTopic
    "<q>Yes, I think that's the diamond.</q> she nods eagerly, <q>Have you tried
whether it fits the ring?</q>"
    isActive = gRevealed('ring-shown')
;

++ GiveShowTopic @ring
    "{The sarah/she} nods eagerly, <q>Yes, that's my ring!</q> she declares, but then
her hand flies to her mouth, <q>But - oh my goodness - the diamond is missing!</q>
    <.reveal ring-shown>"
;

+++ AltTopic
    "<q>That's my ring, all right!</q> {the sarah/she} declares, <q>But you haven't
fitted the diamond!</q>"
    isActive = gRevealed('diamond-shown')
;
```

The sequence of responses if the player shows first the ring, then the diamond, and then the reassembled ring to Sarah (and then finally gives her the ring) may not be absolutely perfect, but it's now starting to get within the bounds of the tolerable, and improving on it further can be left as yet another exercise for the dedicated reader: the first step would probably be to replace the single responses with EventLists; you might also want to test gRevealed('ring-shown') and gRevealed('diamond-shown') on the diamondRing GiveTopic and ShowTopic.

TADS 3 Tour Guide

A little way back we provided a slightly complex GiveTopic to allow the player character to hand Sarah a gas mask. We need her to accept one gas mask, but not both of them, so it would be good to provide an AltTopic that makes Sarah decline the second gas mask if she already has the first. We can achieve this by checking whether anything in Sarah's inventory is of class GasMask, but to do that we need to make isActive a slightly more complex method than normal:

```
++ AltTopic
  "<q>No thanks,</q> she declines, <q>I think one gas mask's enough for anyone,
  don't you?</q> "
  isActive
  {
    foreach(local obj in getActor.contents)
      if(obj.ofKind(GasMask))
        return true;
    return nil;
  }
;
```

Obviously, this should go straight after the relevant GiveTopic.

19.7.6. InitiateTopic

You'll recall a couple of sections back that we ended up with rather a lengthy beforeTravel method on curatorWatching : ConversationReadyState. Although it isn't *too* bad, you may feel that its combination of switch statement and if statements is not only rather cumbersome but out of tune with the general approach of controlling behaviour by defining objects. Well, in fact, TADS 3 does provide a mechanism we could use to implement this case in a radically different way. Here's how the same curator behaviour can be implemented using InitiateTopics:

```
++ curatorWatching : ConversationReadyState
  stateDesc = "He's watching you carefully. "
  specialDesc { inherited; stateDesc; }
  isInitState = true
  beforeTravel(traveler, connector)
  {
    inherited(traveler, connector);
    if(traveler == gPlayerChar)
    {
      initiateTopic(connector);
    }
  }
;

+++ InitiateTopic @museum
  topicResponse
  {
    "{The curator/he} stops you, asking, <q>May I see your
    ticket please?</q><.p><<gSetKnown(museumTicket)>>";
    exit;
  }
;

++++ AltTopic
  topicResponse
  {
    getActor.moveIntoForTravel(byCases);
    "{The curator/he} follows you into the museum. ";
  }
  isActive = (gRevealed('ticket-shown'))
;

+++ InitiateTopic @museumLobby
  topicResponse
  {
    getActor.moveIntoForTravel(behindTable);
    "{The curator/he} follows you into the lobby. ";
  }
;
```

TADS 3 Tour Guide

```
+++ InitiateTopic @benefactorsRoom
    topicResponse
    {
        "<q>I'm afraid you can't go in there,</q> {the curator/he}
        intercepts you, <q>That's the <i>benefactors'</i> room; only
        our benefactors are allowed in there to see the special
        exhibits.</q><.reveal benefactors> ";
        exit;
    }
;

++++ AltTopic
    topicResponse
    {
        foreach(local cur in gPlayerChar.contents)
            if(cur.ofKind(Container) && cur != cap && !cur.ofKind(Matchbook))
            {
                "<q>Sorry,</q> {the curator/he} apologizes as he intercepts
                you, <q>But we can't allow you to take any bags or containers
                in there. It's policy, I'm afraid - one can't be too careful
                these days.</q> ";
                exit;
            }
    }
    isActive = (carbuncle.isIn(getActor))
;
;
```

An `InitiateTopic` is thus a kind of `TopicEntry` that can be used to make an actor initiate conversation (or any other kind of action) in response to something in the environment. Although `InitiateTopics` are set up in exactly the same way as `GiveTopics` and `ShowTopics`, and, just like them, match game objects, they are activated not in response to player conversational commands such as `GIVE TO`, `SHOW TO` or `ASK ABOUT`, but under program control in response to a call to **`initiateTopic(obj)`**. Note that this may be called either on an `Actor` or an `ActorState`, and that it will reference the `InitiateTopics` nested directly within the actor or `ActorState` object on which `initiateTopic` is called.

The example we have just given shows that we can generally use an `initiateTopic(obj)` statement to replace a `switch(obj)` statement and transform the cases within the body of the `switch` statement into individual `InitiateTopics`. At the same time the use of `AltTopic` and `isActive` can be used to replace the `IF... THEN... ELSE...` logic. The only limitation is that the `InitiateTopics` must be keyed on some set of objects (the `obj` parameter of the `initiateTopic` method). These could even be special objects created for the purpose, but more normally they will be existing objects that already serve some purpose in the game. In the most common case, they will often be locations (`Rooms`).

`InitiateTopics` keyed by location are probably most useful in connection with sidekick characters, like Sarah in the present game, who can be made to offer spontaneous observations on (some of) the various locations she visits in company with the player character. The best place to put the call to `initiateTopic()` is probably in the `arrivingTurn()` method of the `sarahFollowingState`, so that it's called each time Sarah arrives in a new location (but not every turn she remains there). Also, the player won't be impressed if Sarah makes the same remark every time she arrives in a given location, so we'll need to provide a list of responses for each case we implement. If we want Sarah to make a remark only on her first visit or two, the easiest way may be to provide an `Event` list with only one or two elements:

```
+ sarahFollowing : AccompanyingState
    specialDesc = "{The sarah/she} is standing beside you. "
    stateDesc = "She's standing beside you. "
    accompanyTravel(leadActor, conn)
    { return leadActor == gPlayerChar; }
    afterAction()
    {
        if(gPlayerChar.isIn(snowMobile))
        {
            getActor.setCurState(sarahOnSnowMobile);
            nestedActorAction(getActor, SitOn, snowMobile);
            "{The sarah/she} gets on the snowmobile behind you. ";
        }
    }
    getAccompanyingTravelState(leadActor, conn)
    { return new SarahInTravelState(location, leadActor, self); }
    arrivingTurn() { initiateTopic(getActor.location); }
;
;
```

TADS 3 Tour Guide

```
++ InitiateTopic, EventList @portDeck
[ '<q>What a big ship!</q> {the sarah/she} remarks. ' ]
;

++ InitiateTopic, ShuffledEventList @tardisControlRoom
[ '<q>Cor! This is impressive!</q> {the sarah/she} remarks. ',
  '<q>Well, here we are again!</q> says {the sarah/she} brightly. ' ]

[ '<q>Well, where shall we go next?</q> {the sarah/she} wonders. ',
  '{The sarah/she} wanders over to the console and inspects the setting
  of the slider and dial. ', nil ]

;

++ InitiateTopic, EventList @snowWorld
[ '<q>Brrr. This place is so <i>cold</i>,</q> {the sarah/she} complains. ',
  '<q>Can\'t we just go back to the Tardis?</q> {the sarah/she} asks. ' ]
;

++ InitiateTopic, EventList @insideHut
[ '{The sarah/she} walks over to the snowmobile and lays her hand on one of
  the seats. Then she turns to you and asks, <q>What do you think?</q> ' ]
;

++ InitiateTopic, EventList @neForestPath
[ '{The sarah/she} starts scuffling the ground with her feet, <q>I think there
  may be something here.</q> she announces. ' ]
isActive = (!woodenTablet.discovered)
;

++ InitiateTopic, EventList @redRavine
[ '<q>That\'s <i>horrible</i>!</q> {the sarah/she} declares, eyeing the
  skeleton with evident disgust. ',
  '<q>What do you think happened to him?</q> she asks, nodding towards
  the skeleton. '
]
;

++ InitiateTopic, EventList @solomonBedroom
[ '{The sarah/she} runs her hand over the bed, then walks over to the window and
  looks out. <q>Hey! Come and have a look at this!</q> she says. ' ]
;

++ InitiateTopic, EventList @museum
[ '{The sarah/she} starts walking round, looking at the exhibits. <q>What
  an odd collection!</q> she declares. ' ]
;

++ InitiateTopic, EventList @graveyard
[ '<q>This place gives me the creeps!</q> {the sarah/she} declares, with a
  little shudder. ',
  '{The sarah/she} walks over to one of the tombstones and examines it
  thoughtfully, <q>Come and have a look at this.</q> she beckons you,
  <q>I think it might be important - like it\'s the key to some sort of
  code or something.</q> '
]
isActive = (!ghost.isIn(graveyard))
;

++ InitiateTopic @outsideCave
"{The sarah/she} looks round at the sky, the valley and the car park, then
turns to you with a slightly melancholy look, <q>It's good to be back out
here,</q> she tells you, <q>but I have this horrible feeling we still have
unfinished business back in <i>there</i>.</q> she nods towards the caves. "
;

+++ AltTopic
topicResponse
{
  "{The sarah/she} turns to you with a broad smile and gives you a great big
```

TADS 3 Tour Guide

```
    hug.\b<q>We <i>did</i> it!</q> she declares, <q>we really did it!</q>\b ";
    endGame(ftVictory);
}
isActive = (goldenBanana.isIn(nil))
;
```

One could probably add more such responses still, but you've probably seen enough to get the general idea, and we're beginning to bring Sarah to life a bit more. Note that our final AltTopic actually ends the game with the player winning - the objective is to get Sarah safely back outside and to destroy the golden banana, which are precisely the conditions under which this AltTopic becomes operative.

Since initiateTopic can key on *any* kind of object, we could also use it to make Sarah respond to SensoryEvents. First, we make her an observer of SoundEvents and SightEvents, then we can start adding suitable InitiateTopics:

```
sarah : Person 'attractive young woman/brunette' 'young woman' @lakeRoom
    "She's an attractive brunette, somewhere in her mid-twenties. "
    isHer = true
    properName = 'Sarah'
    globalParamName = 'sarah'
    knownProp = &sarahKnows
    seenProp = &sarahHasSeen
    notifySightEvent(event, source, info) { initiateTopic(event); }
    notifySoundEvent(event, source, info) { initiateTopic(event); }
;

+ InitiateTopic, EventList @bulkheadOpenEvent
    ['\b<q>That\'s clever!</q> {the sarah/she} declares. ',
     '\b<q>Why not just leave it open?</q> {the sarah/she} wonders. '
    ]
;

+ InitiateTopic, EventList @sailEvent
    ['\b<q>So <i>that\'s</i> how you sail this thing!</q> {the sarah/she} declares. ',
     '\b<q>Here we go again!</q> {the sarah/she} remarks cheerfully. ',
     '\b{The sarah/she} starts whistling <i>A Life on the Ocean Wave</i>. '
    ]
;

+ InitiateTopic @explosionEvent
    topicResponse
    {
        callWithSenseContext( getActor, sight,
            { : "\b<q>Crikey, that sounded close!</q> cries {the sarah/she}. " } );
    }
;

+ InitiateTopic @ghostAppearingEvent
    "\b<q>Eek!</q> cries {the sarah/she}, clutching at your arm.
    <<getActor.setHasSeen(ghost)>><<getActor.setKnownAbout(goldenBanana)>>"
;

;
```

Note that in the last of these we call getActor.setHasSeen(ghost) to record the fact that Sarah has actually seen the ghost. Clearly this will be rather important if we want to define Sarah's responses to conversation about the ghost! At the same time we can note the fact that Sarah has now heard about the golden banana.

For any of these InitiateTopics to work, we obviously have to add some code to define the relevant events and trigger them, first the opening of the bulkhead door:

```
+ bulkheadDoor : HiddenDoor 'bulkhead door/doorway/opening' 'bulkhead door'
    "The central section of the forward bulkhead has slid open, revealing a
    doorway through the bulkhead. "
    destination = crewQuarters
    makeOpen(stat)
    {
        inherited(stat);
        if(stat) bulkheadOpenEvent.triggerEvent(self);
    }
;

bulkheadOpenEvent : SightEvent;
```

TADS 3 Tour Guide

For the sailing of the ship we likewise define `sailEvent : SightEvent`; and then add `sailEvent.triggerEvent(self);` after `ship.moveInto(wheel.curDestination);` on the `SpringLever` object attached to the panel on the quarterdeck. The `explosionEvent` has already been defined (when the bomb detonates on the London street), but note that in order for Sarah's response to it to be displayed we have to use `callWithSenseContext` since the triggering of `explosionEvent` takes place within a `senseFuse`.

Finally, we need to set up the `ghostAppearingEvent`:

```
RoomDaemon
{
    inherited;
    if(!ghost.moved && !statue.isPulled)
    {
        ghost.moveInto(self);
        "A pale ghost rises slowly from one of the tombs, then turns to you,
        pointing its ghostly finger straight at you. <q>You!</q> it cries,
        <q>Yes you - the disturber of my statue! You are the one who must
        carry out the sacred task! You are the one who must retrieve the
        Golden Banana of Discord and cast it into the fires of Mount Gloom
        before it falls into the hands of the Cabal!</q> ";
        ghostAppearingEvent.triggerEvent(ghost);
        gSetKnown(goldenBanana);
        gPlayerChar.setHasSeen(ghost);
    }
}
```

And of course, you need to define:

```
ghostAppearingEvent : SightEvent;
```

Used with sufficient ingenuity, and in combination with `SensoryEvents` or keyed on locations etc, `InitiateTopics` can thus be used to define NPCs' reactions to just about anything we like.

Finally, although in this example we have used `SensoryEvents` to trigger `InitiateTopics` directly on the actor, if we'd really wanted a different set of `InitiateTopics` to handle different responses to these events on each `ActorState`, we could have defined:

```
notifySightEvent(event, source, info) { curState.initiateTopic(event); }
notifySoundEvent(event, source, info) { curState.initiateTopic(event); }
```

19.7.7. AskTopic

An `AskTopic` is used to respond to an `ASK ABOUT` command, e.g. `ASK SARAH ABOUT RING`. Rather than overriding `actionlobjAskAbout` on the `sarah` object, we can nest a number of `AskTopic` objects either in `Sarah` or in one of `Sarah's ActorStates`. To avoid the response becoming repetitive, it's often a good idea to combine an `AskTopic` with some kind of [EventList](#) - often a [StopEventList](#) - which gives several different responses before finally coming to one that's repeated.

One thing a player will often want to ask about an NPC is the NPC him/herself. To handle `ASK SARAH ABOUT HERSELF` we don't need to define some strange kind of `herself` object, we just treat it as equivalent to `ASK SARAH ABOUT SARAH`. Here is how we might define an appropriate `AskTopic` to put in the `sarahTalking` : `InConversationState`.

```
++ AskTopic, StopEventList @sarah
[ { : "<q>What brings you here?</q> you wonder,\b
    <q>I was just taking a walk in the valley when some madman snatched my ring
    off me and ran in here, shouting at me to come and find it if I wanted it
    back.</q> she tells you, <q>So I followed him in. Now I
    just want to find my ring and get out of here.</q><<gSetKnown(diamondRing)>>" },
  { : "<q>And you are...?</q> you ask.\b
    <q><<sarah.makeProper>>,</q> she tells you, <q>My name's Sarah. Pleased
    to meet you - and I'll be even more pleased if you help me find my ring.</q>" },
  { : "<q>So you you've never been in these caves before?</q> you ask.\b
    <q>No, never,</q> she replies, <q>and I wouldn't be here now if my ring
```


TADS 3 Tour Guide

```
    hadn\'t been stolen. Caves aren\'t really my thing, though these
    aren\'t at all what I expected!</q> she nods towards the lake and the
    ship, and a pensive expression crosses her face, <q>I suppose it might
    be interesting to explore - but I really should get back.</q>',
    '<q>What are you so anxious to get back to?</q> you ask.\b
    <q>Fresh air and open sky, for a start!</q> she laughs. '
  ]
;
```

Note that Sarah's first response mentions her ring. Up to that point the player knows nothing about the ring, and so cannot refer to it in conversation; even if a TopicEntry is defined that matches the ring, it won't be activated until the player knows about the ring. One way the player can get to know about the ring is by the player character actually seeing it, but another is through being told about it as here. In order to achieve this we use the `gSetKnown(diamondRing)` macro, which actually translated into `gPlayerChar.setKnowsAbout(obj)`. Likewise, Sarah's second response reveals her name, so we use the custom `makeProper` method both to make the appropriate changes on the Sarah object and to return her proper name. Both methods can be called using the `<<>>` syntax within a double-quoted string, but we can't use a double-quoted string directly in an EventList. We can, however, use a function pointer, and that function pointer can be returned from an anonymous function declaration, and we can use the shortform syntax to declare it. Thus we can use `{ "Here's a double-quoted string" }` as a shorthand way of writing `new Function { "Here's a double-quoted string"; }`. Note that in the short form we *don't* put a semi-colon after the closing double-quote mark.

Since Sarah's answer clearly invites a question about the ring, this is the next thing we should cater for:

```
++ AskTopic, StopEventList @diamondRing
[ '<q>What does your ring look like?</q> you ask.\b
  <q>It\'s a plain platinum band with a solitaire diamond.</q> she tells you. ',

  '<q>This ring - it\'s important to you?</q> you inquire.\b
  <q>Oh yes!</q> {the sarah/she} declares, <q>It\'s not it\'s monetary value, so
  much; it\'s more a sentimental thing - you can\'t replace that with insurance
  money. Besides, I was so <i>incensed</i> when this bloke snatched it off me -
  why should he get away with it?</q>',

  new function {
    "<q>Have you any idea where your ring might be?</q> you ask.\b
    <q>Perhaps he dropped it back in there,</q> she suggests, nodding towards
    the open door,<q>let's go and see.</q><.p>";
    sarah.setCurState(sarahGuidel);
  },

  '<q>And the missing ring...</q> you begin.\b
  <q>... is a plain platinum band with a solitaire diamond.</q> she reminds you. '
]
;
```

Note the new function in the event list; this finally sets Sarah off on her brief [guided tour](#), from which she'll arrive back where she started having accomplished nothing.

The player may also ask Sarah about herself when she starts following the player character around, so we need to put an appropriate AskTopic under the sarahFollowing state. We also need to cater for the possibility that the player has not yet discovered Sarah's name:

```
+ sarahFollowing : AccompanyingState
...
;

++ AskTopic, StopEventList @sarah
[ '<q>So, what you do when you\'re not looking for missing rings?</q> you ask.\b
  <q>Oh, nothing important,</q> she replies, <q>I\'m just a freelance journalist.</q>',
  '<q>Have you come up with many interesting scoops?</q> you wonder.\b
  <q>Now <i>that</i> would be telling.</q> she replies mysteriously. '
]
;

+++ AltTopic
  "<q>What's your name, by the way?</q> you wonder.\b
  <q>I'm <<getActor.makeProper>>,</q> she tells you with a warm smile. "
  isActive = (!sarah.isProperName)
;
```

TADS 3 Tour Guide

Something else the player may ask about is the ghost. It doesn't make sense to discuss the ghost unless both Sarah and the player character have encountered it. The library will take care of putting a ghost topic out of reach until the player character knows about the ghost, but we will want to add an `isActive` condition to make the topic unreachable unless Sarah has also seen it. Moreover, we'll want to allow for the fact that the way the ghost is discussed is likely to be rather different if it's still present at the time, which we'll do by means of an `AltTopic`. Again, we nest all this in the `sarahFollowing` state (since in order to have seen the ghost, Sarah must be following the player character).

```
++ AskTopic, StopEventList @ghost
[
  '<q>What did you make of that ghost?</q> you ask.\b
  <q>Scary,</q> she announces, <q>Definitely scary. But I suppose I might be
  a bit upset if someone had just pushed my statue over - especially if I was
  dead.</q>',
  '<q>So you think it definitely was a ghost we saw in the churchyard?</q> you enquire.\b
  <q>Definitely.</q> {the sarah/she} nods vigorously, <q>Didn\'t you see the way it
  just <i>appeared</i> from that tomb - and then vanished away again?</q>\b',
  '<q>What that ghost said - about a sacred task and a golden banana. Is that something
  we should take seriously, do you think?</q> you ask.\b
  <q>Something <i>we</i> should take seriously?</q> {the sarah/she} replies, cocking
  one eyebrow, <q>It was <i>you</i> he was talking to!</q> Then, seeing your look
  of despair, she gives a little laugh and continues, <q>Come on, then, let\'s go and
  find that banana - we can\'t have you being haunted by a toppled statue for the
  rest of your life!</q>',
  '<q>So you think we\'d better take the ghost seriously.</q> you conclude.\b
  <q>Yes,</q> she says, <q>I do.</q> '
]
isActive = (getActor.hasSeen(ghost))
;

+++ AltTopic
"<q>Is that really what I think it is?</q> you whisper.\b
<q>Yes!</q> she whispers back."
isActive = (ghost.isIn(getActor.location))
;
```

We might also want to cover the case in which a sceptical Sarah has been told about the ghost but hasn't seen it for herself (how that comes about will be dealt with in more detail when we come look at examples of [TellTopic](#)). In this case we can nest the `AskTopic` inside the `sarah` object instead of one of her `ActorStates`, since we'll want this group of replies to be used whichever `ActorState` Sarah happens to be in. The player character (PC) could sail off in the ship, encounter the ghost, then return to tell and ask Sarah about it, all without giving Sarah the ring that makes her follow him, or he could give Sarah the ring between encountering the ghost and trying to discuss it with Sarah.

```
+ AskTopic, ShuffledEventList @ghost

[ '<q>You don\'t believe it was a ghost I saw, do you?</q> you ask.\b
  <q>No - of course not!</q> {the sarah/she} chuckles, <q>I mean, a
  ghost! They don\'t exist - every sensible person knows that!</q>' ]

[ '<q>So you think I\'m lying about the ghost?</q> you ask.\b
  <q>No,</q> she replies carefully, <q>But I think you must be mistaken.</q> ',
  '<q>So, what do you think I saw?</q> you demand.\b
  <q>A trick of the light maybe?</q> she suggests, <q>Or maybe you ate
  too much green cheese last night? Or graveyards bring out the over-
  active imagination in you?</q>',
  '<q>If it wasn\'t a ghost I saw, who or what was it told me about
  the golden banana and the cabal?</q> you ask.\b
  <q>A ghost that burbles about bananas and cabals?</q> she retorts,
  <q>Oh come on! You\'re just having me on!</q>',
  '<q>I\'m sure I did see a ghost, you know.</q> you insist, <q>Do I
  strike you as the sort of person who imagines or invents things?</q>\b
  <q>Not normally, perhaps.</q> {the sarah/she} concedes, <q>Perhaps
  someone was playing a trick on you? A holographic projection? A prankster
  in a sheet? This whole place seems a bit tricky to me!</q>'
]
isActive = (getActor.knowsAbout(ghost))
;
```

Once again we apply an `isActive` condition; this discussion can only take place if Sarah knows about the ghost (because the player character has told her of his encounter with it). Note that if Sarah has actually seen the ghost for

TADS 3 Tour Guide

herself, the responses we defined previously for that eventuality will automatically override those defined just above if Sarah has seen the ghost, since a TopicEntry (even if it's only a DefaultTopic) in the current ActorState always takes precedence over one in the Actor.

So far our AskTopics have all concerned single game objects, but we can also create topics which can be asked (or told) about. For example, we might want to ask the curator about the museum, so we could create a suitable topic object:

```
tMuseum : Topic 'museum';
```

Furthermore, we might want to make asking the curator about the museum and asking him about himself effectively the same. So instead of defining an AskTopic with a single object referenced by @ in its template, we can create it with a list of objects the AskTopic will match:

```
++ AskTopic, StopEventList [curator, tMuseum]
[ { : "<q>Are you in charge here?</q> you ask.\b
    <q>That's right, I'm <<curator.makeProper>>, the curator of this
    museum.</q> he replies with an evident air of self-importance. " },
  '<q>Have you been curator here long?</q> you wonder.\b
    <q>Only for the last twenty years.</q> he replies. ',
  '<q>Do you get many visitors to this museum?</q> you wonder.\b
    <q>One or two.</q> he replies, <q>After all, it\'s not as if many
    people live round here.</q>',
  '<q>Are you interested in acquiring more exhibits?</q> you enquire.\b
    <q>Naturllich - if they are of the right sort.</q> he tells you. '
]
;
```

Clearly, this AskTopic must be nested inside the curatorTalking state, not inside the tMuseum topic object, so be careful where you position it in your code! The same applies to the following:

```
++ AskTopic, StopEventList @tBenefactor
[
  '<q>Who are the benefactors?</q> you ask.\b
    <q>Why, people who have been especially generous to the museum,
    of course.</q> {the curator/he} tells you, in the tone of one
    explaining the absolutely blindingly obvious. ',

  '<q>So how does one become a benefactor?</q> you wonder.\b
    <q>Well,</q> he says, <q>one way would be to procure our most
    wanted acquisition of the month for us.</q>',

  '<q>So if I wanted to become a benefactor...</q> you begin,\b
    <q>Just bring us the exhibit we are most keen to acquire
    just now.</q> {the curator/he} reminds you, <q>We publish
    the details on our monthly flyer.</q>'
]
isActive = gRevealed('benefactors')
;
```

For which we also need to define:

```
tBenefactor : Topic 'benefactor/benefactors';
```

19.7.8. TellTopic

A TellTopic is just like an AskTopic, except that it handle TELL SOMEONE ABOUT SOMETHING instead of ASK SOMEONE ABOUT SOMETHING. You'll recall that when we were defining AskTopics for Sarah, we allowed the possibility that she might learn of the ghost either by seeing it herself, or by the player character reporting seeing it. So the obvious example of a TellTopic would be to allow the player character to report seeing the ghost to Sarah, who'll be pretty sceptical about it:

```
+ TellTopic, StopEventList @ghost
[
  /* In the first response we call the appropriate method to note that
    Sarah has now heard about the ghost */
]
```

TADS 3 Tour Guide

```
{: "<q>Do you know what? I saw a ghost! Over in the graveyard on the west shore
of the lake!</q> you tell her. <<sarah.setKnowsAbout(ghost)>>\b
<q>A ghost!</q> she cries derisively, <q>Oh, come on!</q>" },

/* Likewise in the second response we note that Sarah has now heard of the
golden banana */

{: "<q>I really did see a ghost.</q> you insist, <q>After I pushed its statue
over. It wants me to find a golden banana before the cabal gets hold of
it!</q> <<sarah.setKnowsAbout(goldenBanana)>>\b
<q>What have you been drinking?</q> {the sarah/she} enquires, <q>Ghost? Golden
banana? Cabal?</q> she shakes her head and laughs. " },

'<q>I <i>did</i> see a ghost,</q> you repeat.\b
<q>Yes, yes, and I suppose it was rattling its chains and moaning in
a hollow voice.</q> she mocks.',

'<q>I\'m absolutely positive I saw a ghost.</q> you insist.\b
In reply, she merely looks at you and shakes her head in mock despair. '
]
;

/* If Sarah has actually seen the ghost her response will be rather different */

++ AltTopic
"<q>I saw a ghost...</q> you begin.\b
<q>I know, I was there - remember?</q> she replies. "
isActive = (getActor.hasSeen(ghost))
;

/* Finally, and overridingly, the conversation will go rather different if the
ghost is actually present */

++ AltTopic
"<q>That's a ghost, isn't it?</q> you remark, pointing at the phantom.\b
<q>A ghost, yes, definitely a ghost.</q> {the sarah/she} concurs in a
very small voice. "
isActive = (ghost.isIn(getActor.location))
;
```

Once again we nest this TellTopic directly in Sarah, since it could come up either before or after the player character has handed over the ring, and hence before or after Sarah has started following the player character around. Note that when we get to the AltTopics we list them in reverse order of precedence, that is the lower down the list the AltTopic comes the higher will be its precedence if its isActive property is true.

19.7.9. AskTellTopic

An AskTellTopic is effectively a combination AskTopic and TellTopic; in other words it responds to both ASK X ABOUT Y and TELL X ABOUT Y, treating the two just the same. It is thus useful when you want players to be able to talk about a topic, but feel it doesn't make much difference whether they do so by telling or asking.

Let's suppose that this is how we want to handle conversing with the demons about the golden banana. We could put the following into the demonsChattering state:

```
++ AskTellTopic, StopEventList @goldenBanana
[
'<q>Say, do you little devils know anything about a Golden Banana?</q> you enquire.\b
<q>The Golden Banana!</q> one of them exclaims, <q>He knows of the Banana!
We wants it! We wants it!</q> ',
'<q>You seem interested in this Banana.</q> you remark.\b
The chief demon draws himself up to his full height and stares straight
at your navel. <q>Know that I am Princifax, leader of demons, Lord High
Big Stink, and recipient of three Rotten Tomato awards for Ham Actor of
the Underworld. Fear me! Fear me!</q>\b
<q>All hail Lord High Big Stick Princifax of the Rotten Tomato!</q> chorus
the other demons.\b
```

TADS 3 Tour Guide

```
<q>Know too, feeble mortal, that the banana of which you speak is OURS!  
We <i>demand</i> that if you know anything of it, you bring it to us.  
Otherwise...</q> Princifax shakes his head, as if contemplating a fate worse  
than death for you. ',  
'<q>Why\'s this banana so important to you?</q> you wonder.\b  
<q>It was stolen from us,</q> the demons reply indignantly, <q>By one of  
your kind - so that makes <i>you</i> responsible for bringing it back to  
us.</q>',  
'<q>So you\'d like me to bring you back this banana?</q> you surmise.\b  
<q>For a human you\'re almost quick on the uptake.</q> one of them  
replies.'  
]  
;
```

Another reason for using a combination AskTellTopic might be when the command the player is most likely to use might alternate over the course of the topic as the situation changes. For example, depending on whether the player character has yet purchased a museum ticket or shown it to the curator, he might ask where to purchase a ticket, tell the curator that he's already purchased one, or ask whether he needs to buy another if he wants to return. We could handle this with the following group of TopicEntries, which should go in the curatorTalking InConversationState:

```
++ AskTellTopic @museumTicket  
  "<q>How do I get a museum ticket?</q> you want to know.\b  
  <q>There's a vending machine down the hall.</q> he tells you,  
  <q>You can get one there.</q>"  
;  
  
+++ AltTopic  
  "<q>I have bought a ticket.</q> you assure him.\b  
  <q>Let's see it, then.</q> {the curator/he} insists."  
  isActive = (museumTicket.moved)  
;  
  
+++ AltTopic  
  "<q>Do I need another ticket if I want to come in again?</q>  
  you ask.\b  
  <q>No,</q> he assures you, <q>the ticket you showed me is  
  good for multiple entry.</q> "  
  isActive = gRevealed('ticket-shown')  
;
```

19.7.10. AskForTopic

Just as an AskTopic deals with commands like ASK FRED ABOUT BEER so an AskForTopic handles commands like ASK FRED FOR BEER. The programming principles are just the same - but the game logic may be somewhat more complex. For one thing, as well as asking for game objects like the ring or a coin, the player could try asking NPCs for abstract things like advice or directions. For another, there are nearly always more variables to consider. If I ASK SARAH FOR ADVICE what advice she can relevantly give would very much depend on the state of play at the time, and might be very difficult to implement decently across the whole game. If I ASK SARAH FOR RING then how she responds will depend, not only on her willingness or otherwise to part with the ring, but also on whether she has the ring, or whether the PC has the ring, and maybe other things besides. It's up to us, as the game author, to handle all this.

Sarah's ring may be a good example to start with, since it's relatively straightforward. We know that once Sarah has got her ring she switches into the sarahFollowing state, and we'll assume that she isn't too keen to part with it thereafter. Therefore, if we put an AskForTopic @diamondRing in sarahFollowing, we can safely assume that Sarah does actually have the ring.

The reason the player character may want to request the ring back from Sarah is that the diamond is needed to cut open a couple of glass objects, the glass jar containing the blue crystal and the display case containing the golden banana. The PC may or may not have cut open the first of these before he hands the ring over to Sarah. If he then decides he needs the diamond to cut open the jar, he may try asking Sarah for the ring back. Sarah won't give it - but there's no reason why Sarah shouldn't do the cutting herself. So we'll assume that if the glass jar is accessible and not yet cut open when the PC asks Sarah for her ring, he'll explain that he needs it to cut open the jar and she duly obliges by doing it herself:

TADS 3 Tour Guide

```
++ AskForTopic @diamondRing
  "<q>Can I borrow your ring a moment, please?</q> you ask.\b
  <q>Whatever for, I've only just got it back!</q> {the sarah/she}
  complains. "
;

+++ AltTopic
  topicResponse
  {
    "<q>Can I borrow your diamond ring a moment, please?</q> you ask.
    <q>I need something hard to cut this glass jar open.</q>\b
    <q>Here, let me.</q> {the sarah/she} replies, taking the jar. ";
    glassJar.moveTo(getActor);
    newActorAction(getActor, CutWith, glassJar, diamondRing);
    "<q>Here you are then,</q> she declares, handing the now
    opened glass jar to you. ";
    glassJar.moveTo(gPlayerChar);
  }
  isActive = (getActor.canTouch(glassJar) && !glassJar.isOpen)
;
```

We'll handle cutting open the glass display case later, when we come to look at [ConvNode](#). For now we'll proceed with a different example, the root and the carbuncle we need from Solomon. First we need to make Solomon prevent the PC from taking either the root or the carbuncle unless he's given permission for them to be taken. We'll add a gifted property to both the root and the carbuncle (add a line saying `gifted = nil` to the definition of both carbuncle and baarasRoot) and then override Solomon's `beforeAction()` method (we override it on solomon rather than one of his ActorStates since he could be in any of his ActorStates when the PC attempts to take the root or the carbuncle):

```
solomon : Person 'middle-aged middle aged man' 'middle-aged man' @solomonChair
  "He's quite good-looking in a middle-eastern sort of way, with long curly
  black hair that's just starting to go grey, and a neatly kept beard. He's
  dressed in a purple cloak. "
  isHim = true
  posture = sitting
  properName = 'King Solomon'
  globalParamName = 'solomon'
  beforeAction()
  {
    inherited;
    if(gActionIs(Take) || gActionIs(TakeFrom))
    {
      if(gDobj is in (baarasRoot, carbuncle) && !gDobj.gifted)
      {
        "As you reach out to take {the dobj/him}, {the solomon/he} calmly
        but firmly interrupts you.
        <q>You do not take a king's property without his permission.</q>
        he tells you sternly. ";
        exit;
      }
      if(gDobj == bronzeBowl && gRevealed('bowl-returned'))
      {
        "Before you can touch the bowl, {the solomon/he} remarks, <q>Since
        I've offered to reward you for returning that bowl, I think you
        had better leave it alone.</q> ";
        exit;
      }
    }
  }
;
```

For both the root and the carbuncle, we want an AskForTopic that causes the request to be denied until the PC has given Solomon his bowl back, and regarded as superfluous once the root or carbuncle has been taken. Here's how we might do it for the root (nesting the AskForTopic inside the solomonTalking state):

```
++ AskForTopic @baarasRoot
  "<q>May I have that strange looking root?</q> you ask.\b
  <q>No, I'm studying it.</q> he replies. "
;
```

TADS 3 Tour Guide

```
+++ AltTopic
  topicResponse
  {
    "<q>May I have that root you've got there?</q> you ask.\b
    <q>Help yourself.</q> he replies. ";
    baarasRoot.gifted = true;
  }
  isActive = gRevealed('bowl-returned')
;

+++ AltTopic
  "<q>May I take that root?</q> you request.\b
  <q>You already have.</q> {the solomon/he} points out. "
  isActive = (baarasRoot.moved)
;
```

This is fine, except that we have to repeat almost exactly the same thing for the carbuncle. This may not be too bad here, but for multiple objects it could become tedious, so it may be worth exploring an alternative approach. For this we need to abandon the nice neat declarative syntax of AltTopics and revert to the bad old ways of if and else statements:

/ Approach 2 - messier but more concise for multiple objects: */*

```
++ AskForTopic [baarasRoot, carbuncle]
  handleTopic(fromActor, topic)
  {
    /* An AskForTopic matches a ResolvedTopic - to get back to the object
       match we need to use getBestMatch */

    local obj = topic.getBestMatch();

    /* Set a message parameter for convenience */
    gMessageParams(obj);

    /* Start with a question common to all situations */
    "<q>May I take {the obj/him}?</q> you ask.\b";

    /* Handle Solomon's response according to the state of play */
    if(obj.moved)
      "<q>You already have.</q> {the solomon/he} points out. ";
    else if(gRevealed('bowl-returned'))
    {
      "<q>Help yourself,</q> {the solomon/he} offers with an expansive
      wave of the hand, <q>Is there anything else you'd like?</q> ";
      obj.gifted = true;
    }
    else
      "<q>No, I'm studying it.</q> he replies. ";
  }
;
```

It is less immediately obvious how this works, but it's not too bad, and certainly more compact than writing out a second AskForTopic with its associated pair of AltTopics, and has the merit that it could easily be extended to a large number of objects simply by adding them to the match list at the start of the object declaration. One downside, though, is that the exchange between Solomon and the PC is rigidly stereotyped.

A third possibility is to define a special class that handles requests for rewards from Solomon, allow some variety and customization within the class, and then instantiate the class for each item we want it to handle:

/ Approach 3 - perhaps the best compromise for several similar cases */*

```
class AskForRewardTopic : AskForTopic
  topicResponse
  {
    /* We can use the matchObj property to get at the object this topic matches,
       provided we always define it to match one and only one */

    /* First we get the player character to ask for the object */
    requestQuestions.obj = matchObj;
    requestQuestions.doScript;
```

TADS 3 Tour Guide

```
/* If the matchObj has moved the player character has already taken it */
    if(matchObj.moved)
        "<q>You already have.</q> {the solomon/he} points out. ";

/* Otherwise, if the bowl has been returned, grant the request */
    else if(gRevealed('bowl-returned'))
    {
        grantRequest;
        matchObj.gifted = true;
    }
/* Otherwise, refuse the request */
    else
        refuseRequest;
}
requestQuestions : ShuffledEventList
{
    [
        '<q>May I take '+ obj.theName +', please?</q> you ask.\b' ,
        '<q>Could I have ' + obj.theName +'?</q> you request.\b' ,
        '<q>I\'d like to take ' + obj.theName +', is that okay?</q> you ask.\b',
        '<q>That '+ obj.name + ' looks interesting, may I take it?</q> you ask.\b'
    ]
    obj = nil
}
refuseRequest = "<q>No, I'm studying it.</q> he replies.<.p>"
grantRequest = "<q>Help yourself,</q> {the solomon/he} offers with an expansive
    wave of the hand, <q>Is there anything else you'd like?</q><.p>"
;

/* Then add an AskForRewardTopic for each of the possible rewards: */

++ rootTopic : AskForRewardTopic @baarasRoot
;

++ AskForRewardTopic @carbuncle
refuseRequest = "<q>No, it was a personal gift from the Queen of Sheba
    and I see no reason why you should have it.</q> he replies.<.p>"
grantRequest = "<q>Very well, take it,</q> {the solomon/he} sighs, <q>It\'s
    only a pretty bauble, after all.</q><.p>"
;
```

19.7.11. AskAboutForTopic

An AskForAboutTopic handles both ASK X FOR Y and ASK X ABOUT Y as if they meant the same thing. For example, since King Solomon was famed for his wisdom, one might ASK SOLOMON FOR WISDOM or ASK SOLOMON ABOUT WISDOM and reasonably receive the same kind of answer:

```
++ AskAboutForTopic, ShuffledEventList @tWisdom
[
    '<q>The fear of the Lord is the beginning of wisdom.</q> he tells you,
    <q>and the knowledge of the Holy One is insight.</q> ',

    '<q>The beginning of wisdom is the most sincere desire for instruction,</q>
    {the solomon/he} declares, <q>and concern for instruction is love of her,
    and love of her is the keeping of her laws, and giving heed to her laws
    is assurance of immortality, and immortality brings one near to God.</q> ',

    '<q>Wisdom is more mobile than any motion; because of her pureness she
    pervades and penetrates all things. For she is a breath of the power of
    God, and a pure emanation of the glory of the Almighty.</q> he replies. ',

    '<q>The wise of heart will heed commandments, but a prating fool will come
    to ruin.</q> he declares. ',

    '<q>Wisdom is better than jewels,</q> he concurs, <q>and all that you
    desire cannot compare with her.</q> ',
```


TADS 3 Tour Guide

```
'<q>The teaching of the wise is a fountain of life,</q> {the solomon/he}
nods sagely, <q>that one may avoid the snares of death.</q> ',

'<q>I prayed and understanding was given me,</q> he tells you, <q>I called
upon God and the spirit of wisdom came to me. I preferred her to sceptres
and thrones, and I accounted wealth as nothing in comparison with her. '
]
;

tWisdom : Topic 'wisdom';
```

One can be quite sure that Solomon never actually uttered any of these quotations from the Biblical books of Proverbs and Wisdom of Solomon - the latter being most probably a first-century (BCE or CE) text - but they seem as suitable as anything here; we are in any case dealing with the Solomon of legend rather than the Solomon of history in this game!

19.7.12. AskTellShowTopic

An AskTellShowTopic, as you might guess, responds to ASK X ABOUT Y, TELL X ABOUT Y or SHOW Y TO X. As an example we could take the ship floating by the shore, which the player character might either ask or tell Sarah about, or else point out to her:

```
+ AskTellShowTopic, StopEventList @ship
[
  '<q>Do you know anything about that ship?</q> you ask.\b
  <q>Not really.</q> she admits, <q>Only that it floats.</q>',

  '<q>How long has that ship been there?</q> you wonder.\b
  <q>Well, it was there when I came, and it hasn\'t moved since.</q>
  she replies. ',

  '<q>Do you think we could use that ship to sail out of here?</q>
  you ask.\b
  <q>I have no idea - why don\'t you go and investigate?</q> she suggests.'
]
;

++ AltTopic
  "<q>What do you make of this ship?</q> you ask.\b
  <q>Let's investigate some more.</q> she suggests,
  <q>How about looking over there?</q> she points vaguely. "
  isActive = (getActor.location.ofKind(Deck))
;

++ AltTopic
  "<q>So, what do you think of the ship?</q> you ask.\b
  <q>You're the one that's sailed it!</q> she points out."
  isActive = (ship.moved)
;
```

Note, however, that when the ship is not explicitly in view (i.e. the ship object is not visible in the current location) SHOW SHIP will not, in fact, behave the same as ASK ABOUT SHIP or TELL ABOUT SHIP. You can verify this by asking SHOWing Sarah the ship while she's standing on the shore, then getting her to follow you aboard and trying ASK SARAH ABOUT SHIP and SHOW SARAH SHIP.

19.7.13. AskTellGiveShowTopic

An AskTellGiveShow topic is one that combines responses to ASK X ABOUT Y, TELL TELL ABOUT Y, GIVE Y TO X and SHOW Y TO X. Having something that responds to such a wide range of commands may not be useful that often, but as an example we'll give Sarah such a topic to respond to news of the discovery of the gold tablet:

```
++ AskTellGiveShowTopic @goldTablet
  "<q>I've found a solid gold tablet!</q> you declare, <q>What do you think
```

TADS 3 Tour Guide

```
it's worth?</q>\b
<q>Cor! It must be worth a fortune!</q> she exclaims. "
;
```

This should be put under the sarahFollowing : AccompanyingState.

Again, if you have Sarah with you when you first topple the statue, note the slightly different behaviours of TELL SARAH ABOUT GOLD TABLET and GIVE SARAH GOLD TABLET; the latter, but not the former, will trigger an implied TAKE command (the same variation will be found between ASK and SHOW). Furthermore, if, having examined the tablet, you leave it where it is and return to the beach, you'll find you can talk to Sarah about the tablet with ASK ABOUT and TELL ABOUT, but you can no longer SHOW or GIVE to her.

Thus, although an AskTellGiveShowTopic responds to all four commands, it does not mean that all four commands will necessarily be treated the same under all circumstances; the object still has to be physically present for GIVE and SHOW to be valid.

19.7.14. Yes,No & SpecialTopics

[YesTopic](#), [NoTopic](#) & [SpecialTopic](#) are really only useful inside [Conversation Nodes](#), so we shall deal with them there.

19.7.15. HelloTopic

A HelloTopic is normally used in connection with [greeting protocols](#). Its main purpose is to respond to a HELLO command or equivalent (such as TALK TO X). The normal place to put a HelloTopic would be in a [ConversationReadyState](#), where it will respond to any conversational command addressed to the NPC just before the NPC is switched into the corresponding [InConversationState](#). If you want to differentiate between the case in which a conversation starts with an explicit HELLO, or TALK TO command, and when it starts with some other conversational command such as ASK ABOUT, you can use an [ImpHelloTopic](#) for the latter.

Since King Solomon initiates the conversation the first time the player character walks in on him (we'll code that later), there's no need to provide a HelloTopic for that initial conversation, but you might want one for subsequent encounters, such as:

```
+++ HelloTopic
  "<q>I have returned, your majesty.</q> you announce.\b
  <q>So I observe.</q> the king replies, looking up at you. "
;
```

The trouble with the above HelloTopic, however, is that it assumes that the player character has been away and returned between conversations, which is not necessarily the case. What we need is a version of the HelloTopic that displays the above message if the player character has been away, but a different message otherwise. So instead of the above, we could define (locating this in solomonExamining):

```
+++ solomonHelloTopic : HelloTopic, StopEventList
[
  '<q>I have returned, your majesty.</q> you announce.\b
  <q>So I observe.</q> the king replies. ',
  '<q>Your majesty,</q> you begin.\b
  <q>Yes?</q> the king replies, turning in his chair and looking
    up at you. '
]
curScriptState = 2
;
```

We set curScriptState to 2 initially, to cater for the possibility that the player strikes up a second conversation with the king before ever leaving his study. We next need to reset curScriptState to 1 each time the player character does leave the king's presence. We can do this by adding the following method to the solomon object itself:

```
solomon : Person 'middle-aged middle aged man' 'middle-aged man' @solomonChair
...
beforeTravel(traveler, connector)
{
  if(connector == solBedroomDoorOutside)
    solomonHelloTopic.curScriptState = 1;
```

TADS 3 Tour Guide

```
    inherited(traveler, connector);  
  }  
;
```

There's one more thing you should know about HelloTopic (and this applies to ImpHelloTopic too). As we've already noted, a HelloTopic is triggered by an explicit greeting *or* by any other conversational command. Internally this means that if any other kind of TopicEntry is triggered when an actor is in a ConversationReadyState, a HelloTopic is triggered. Potentially, this could mean that a HelloTopic tries to trigger itself, which would then have it try to trigger itself, and so on *ad infinitum* (or, in practice, till you got a stack overflow error). The way HelloTopic (and ImpHelloTopic) prevent this is by defining **impliesGreeting** = nil.

19.7.16. ImpHelloTopic

An ImpHelloTopic (if defined) deals with the [greeting protocols](#) in the case where a conversation is started without an explicit greeting command (such as SARAH, HELLO or TALK TO SARAH). The normal place for an ImpHelloTopic is in a [ConversationReadyState](#).

For example, we might provide King Solomon with an ImpHelloTopic that looks like this:

```
+++ ImpHelloTopic  
    "Solomon turns and looks up at you as you start to speak."  
    <<solomonHelloTopic.advanceState()>>"  
;
```

This should be located in the solomonExamining ConversationReadyState. The one subtlety here is the call to solomonHelloTopic.advanceState(). The reason for calling this is that once this ImpHelloTopic has been triggered, we'd want to see the second response, not the first, in the list of responses on [solomonHelloTopic](#).

19.7.17. ByeTopic

A ByeTopic is usually used in connection with [greeting protocols](#), to handle the end of a conversation. As such, it is normally located in the [ConversationReadyState](#) to which the actor returns at the conclusion of the conversation. The ByeTopic handles explicit termination of a conversation (via a BYE command).

In the case of King Solomon, we shall assume that our player character would not be so gauche as to walk out on the king without some attempt at formal leave-taking, so we really want a ByeTopic that handles both implicit and explicit conversation termination. As with the HelloTopic and the ImpHelloTopic this should go in the solomonExamining ConversationReadyState:

```
+++ ByeTopic, ShuffledEventList  
[  
    '<q>Goodbye, your majesty.</q> you say, with a little bow, just to be  
    on the safe side.\b  
<q>Farewell, mysterious messenger,</q> the king replies, <q>perhaps  
    we shall meet again.</q> ',  
    '<q>Well, er, cheerio then, sire.</q> you say.\b  
    Solomon cocks one eyebrow, <q>Cheerio? Is that how angels speak?  
    Fare thee well.</q>',  
    '<q>Goodbye, sir.</q> you bid him farewell.\b  
<q>God be with you too.</q> the king replies formally, before returning  
    to his contemplations. '  
]  
;
```

Note the symmetry here between HelloTopic and ByeTopic: while HelloTopic handles both implicit and explicit conversation initiation, (as of TADS 3.0.6p) ByeTopic handles both explicit and implicit conversation termination.

19.7.18. ImpByeTopic

An ImpByeTopic is yet another kind of [TopicEntry](#) used in [greeting protocols](#). Like [ByeTopic](#), it is used at the end of a conversation, and would normally be placed in a [ConversationReadyState](#). Unlike ByeTopic, it handles only implicit conversation endings, which occur either when the player character simply leaves the area in mid-conversation, or else fails to address a conversational command to the NPC for enough turns to exhaust the NPC's attention span.

For example, we might define the following ImpByeTopic for the curator, which should be located in his curatorWatching ConversationReadyState:

```
+++ ImpByeTopic
    "{The curator/he} turns away and goes back to what
    passes for his work. "
;
```

Note that this doesn't distinguish between the cases where the conversation ends because the PC walks away from the curator and where it ends because the curator becomes bored waiting for us to speak. If we want to make that distinction, we can use [LeaveByeTopic](#) and [BoredByeTopic](#).

19.7.19. LeaveByeTopic

A LeaveByeTopic is effectively a more specialised ImpByeTopic. We can use it in the case where we want to distinguish a conversation ending through the PC walking away from one ending because his interlocutor gives up waiting for him to speak. So, for example, we could add the following, which should also be located in his curatorWatching ConversationReadyState:

```
+++ LeaveByeTopic
    "{The curator/he} watches you leave, and then resumes working. "
;
```

And then we'll see a different message depending on whether the conversation ends as a result of boredom or travel. Note that in this case the (more specific) LeaveByeTopic will be used when the PC departs, and the (less specific) ImpByeTopic will be used when the curator becomes bored, although we could achieve the same effect by using a [BoredByeTopic](#).

19.7.20. BoredByeTopic

The BoredByeTopic is the other special case of [ImpByeTopic](#), and complements [LeaveByeTopic](#). It is triggered whenever a conversation is terminated because an NPC gives up waiting for the PC to speak.

We could, for example, change the curator's ImpByeTopic to a BoredByeTopic:

```
+++ BoredByeTopic
    "{The curator/he} turns away and goes back to what
    passes for his work. "
;
```

In this case the change won't make any practical difference, since either way (given that we've defined a separate LeaveByeTopic) this will be the topic that's invoked when the curator tires of waiting for the PC to talk, although using BoredByeTopic in this case probably makes our source code a little clearer.

19.7.21. ActorByeTopic

ActorByeTopic is the third specialization of ImpByeTopic. It is used when the NPC terminates the conversation of its own accord via npc.endConversation(). In the absence of an active ActorByeTopic, the active ImpByeTopic will be used instead.

npc.endConversation() is used to make an NPC break off a current conversation. This is the complement to initiateConversation; it causes the NPC to effectively say BYE on its own, rather than waiting for the PC to decide to

TADS 3 Tour Guide

end the conversation. This call is mostly useful when the actor's current state is an `InConversationState`, since the main function of this routine is to switch to an out-of-conversation state.

For example, if our curator was particularly cantankerous, we might, for example, tweak one or more of his `TopicEntries` to call `endConversation()`:

```
+++ AltTopic
  "<q>Do I need another ticket if I want to come in again?</q>
  you ask.\b
  <q>No,</q> he assures you, <q>the ticket you showed me is
  good for multiple entry.</q> <<getActor().endConversation()>> "
  isActive = gRevealed('ticket-shown')
;
```

And then add the following in his `ConversationReadyState`:

```
+++ ActorByeTopic
  "<q>Now, if that's all, I have work to do,</q> he adds, turning back to the
  pile of papers on his desk. "
;
```

19.7.22. HelloGoodbyeTopic

A `HelloGoobyTopic` is used for both greeting and farewell. It may be difficult to imagine how the same response could be suitable for both, but perhaps the following (which we'll use for the curator - put it in his [ConversationReadyState](#)) just about does the trick:

```
+++ HelloGoodbyeTopic
  "<q>Good day!</q> you greet him.\b
  <q>Good day to you,</q> {the curator/he} replies. "
;
```

But note: this causes a run-time error in TADS 3.0.6n. The fix (implemented in TADS 3.0.6o) is as follows:

```
modify HelloGoodbyeTopic
  impliesGreeting = nil
;
```

19.7.23. MiscTopic

`MiscTopic` is the parent class of `TopicEntry` types such as [ByeTopic](#), [HelloTopic](#), [YesTopic](#) and [NoTopic](#) that respond to simple command such as HELLO, BYE, YES and NO without referring to any further object or topic (you don't HELLO SARAH ABOUT FISH as you might ASK SARAH ABOUT FISH, for example).

Although you'll never have occasion to use `MiscTopic` directly in the sense of creating `MiscTopics` in your own game, you can use it as the basis of creating subclasses to extend the conversational commands available in your game. For example, suppose you want to create a `PraiseTopic` class that provides a response to commands like PRAISE SARAH or PAY SARAH A COMPLIMENT; there are several steps involved in getting this to work, but it's perfectly feasible to do.

First, simply define `PraiseTopic` as subclass of `MiscTopic`:

```
class PraiseTopic: MiscTopic
  includeInList = [&miscTopics]
  matchList = [praiseTopicObj]
;

praiseTopicObj : object;
```

(Here, `praiseTopicObj` is simply a dummy object that `PraiseTopic` will always match, in default of it's having a `Thing` or `Topic` to match).

TADS 3 Tour Guide

Next define the action and the vocabulary for your new conversation verb:

```
DefineTAction(Praise)
;

VerbRule(Praise)
  (('praise' | 'flatter' | 'compliment') singleDobj)
  | (('pay' | 'offer') singleDobj ('a' |) 'compliment')
  : PraiseAction
  verbPhrase = 'praise/praising (whom) '
;
```

Then provide default handling for this verb on inanimate objects:

```
modify Thing
  dobjFor(Praise)
  {
    verify() { illogical('Praising {a dobj/him} is a waste of breath. '); }
  }
;
```

The slightly more complicated part is defining the handling of the new verb on the Actor class, but this will always follow the same pattern:

```
modify Actor
  dobjFor(Praise)
  {
    preCond = [canTalkToObj]
    verify()
    {
      /* it is vain to praise oneself */
      if (gActor == self)
        illogical('Vanity! Vanity! ');
    }

    action()
    {
      /* note that the issuer is targeting us with conversation */
      gActor.noteConversation(self);

      /* let the state object handle it */
      curState.handleConversation(gActor, praiseTopicObj, praiseConvType);
    }
  }

  defaultPraiseResponse(actor)
  {
    "\^<<theName>> appears totally unmoved by your flattery. ";
  }
;
```

The new defaultPraiseResponse method we've just defined here does what it says it does: it handles any PRAISE commands directed to the actor if they're not dealt with by a PraiseTopic or DefaultAnyTopic. You may be wondering, however, how the game *knows* to use this newly-defined method in such a case. You may be wondering also where the praiseConvType object referred to in the action handler comes from. We can now answer both questions together by defining the praiseConvType object, which is the final piece of the mechanism we need to put in place:

```
praiseConvType : ConvType
  unknownMsgType = 'No-one\'s listening. '
  topicListProp = &miscTopics
  defaultResponseProp = &defaultPraiseResponse
  defaultResponse(db, other, topic)
  { db.defaultPraiseResponse(other); }
;
```

Now we can put our new PraiseTopic to work. We can start by locating the following in sarah's sarahTalking : InConversationState:

TADS 3 Tour Guide

```
++ PraiseTopic, ShuffledEventList
[
  '<q>Has anyone told you what an attractive woman you are?</q> you enquire.\b
  <q>I\'m quite immune to such flattery.</q> she informs you. ',
  '<q>You strike me as a brave and resourceful woman.</q> you remark.\b
  <q>What <i>can</i> you be basing that opinion on?</q> she complains -
  though she does not seem entirely displeased. '
]
[
  '<q>I think I could really get to like you.</q> you tell her.\b
  <q>I\'m still reserving judgment on you.</q> she replies cautiously. ',
  '<q>You\'re a good woman - you know that?</q> you ask.\b
  <q>Less talk, more action.</q> she retorts, <q>It\'s your help
  I need - not your praise.</q> ',
  '<q>You know, I do feel you\'re basically a kind and decent person.</q>
  you tell her.\b
  <q>You\'re too kind.</q> she responds dryly. ',
  '<q>It was really very brave of you to dash in here after that thief
  who stole your ring.</q> you remark, <q>I\'m really very impressed!</q>\b
  <q>Brave? Or just daft?</q> she laughs, <q>I was so incensed I didn\'t
  stop to think about being <i>brave</i>.</q>'
]
isActive = (getActor.isProperName)
;
```

The `isActive` condition here ensures that the player at least gets to know Sarah's name (and has thus asked her a couple of questions about her) before he can start heaping praise on her. We'll enforce the same condition we put on the `PraiseTopic` we'll locate in her `sarahFollowing` state:

```
++ PraiseTopic, StopEventList
[
  '<q>It\'s really very good of you to come with me.</q> you
  tell her.\b
  <q>Not at all,</q> she replies, <q>I\'m just hoping you are
  going to find the way out of here.</q>',
  '<q>I\'m really glad you\'re with me - whatever your reasons.</q>
  you remark, <q>Quite frankly, I\'m glad of your company.</q>\b
  <q>Me too,</q> she admits.',
  '<q>I do like you, you know.</q> you tell her.\b
  <q>Now, don\'t go all sentimental on me!</q> she complains, though
  there\'s a warm twinkle in her eye as she says it.',
  '<q>I\'m afraid I can\'t help finding you very attractive.</q> you
  confess.\b
  <q>I daresay neither of us can help that - what you find attractive,
  I mean.</q> she replies matter-of-factly. ',
  '<q>I really do appreciate having you around.</q> you tell her.\b
  <q>So you keep saying,</q> she observes. '
]
isActive = (getActor.isProperName)
;
```

This is likely to be more interesting if Sarah's relationship with the player character advances so that as circumstances change, so do her responses to his compliments. Perhaps the shared experience of encountering the ghost might be one such event, so we could add:

```
+++ AltTopic, SuggestedPraiseTopic, StopEventList
[
  '<q>You\'ve been pretty brave about that ghost - after all it
  was scary!</q> you remark.\b
  <q>It was,</q> she agrees, <q>and you\'ve been pretty brave too!</q>',

  '<q>You know, I really <i>am</i> glad you\'re with me.</q> you tell her.\b
  <q>Me too!</q> she agrees warmly, giving your arm a little squeeze. '
]
isActive = (getActor.hasSeen(ghost))
;
```

Obviously, this could be taken a lot further, but enough has been said to demonstrate the principle of creating a new kind of `MiscTopic`. One problem remains: how is the player to know that a new kind of conversational command (`PRAISE X`) is available? We'll solve this problem shortly by defining a new kind of [SuggestedTopic](#) to match.

19.7.24. TopicGroup

A TopicGroup is an abstract container for a set of TopicEntry objects. The purpose of the group object is to apply a common "is active" condition to all of the topics within the group.

The isActive condition of the TopicGroup is effectively AND'ed with any other conditions on the nested TopicEntries. In other words, a TopicEntry within the TopicGroup is active if the TopicEntry would otherwise be active AND the TopicGroup is active.

TopicEntry objects are associated with the group via the 'location' property - set the location of the TopicEntry to point to the containing TopicGroup.

You can put a TopicGroup anywhere a TopicEntry could go - directly inside an Actor, inside an ActorState, or within another TopicGroup. The topic entries within a topic group act as though they were directly in the topic group's container.

The TopicGroup is potentially a highly versatile way of organizing TopicEntries when the way you want to organize them doesn't fall neatly into the categories offered by actors, actor states and conversation nodes.

For example, suppose you wanted Sarah to give a different set of responses based on her mood, but all within the same state (the sarahFollowing state, for example). Note that this goes beyond what we're actually attempting in this game, but what you might do is first define some enums for Sarah's moods:

```
enum mHappy, mSad, mAngry;
```

Then you'd need to define a mood property, say, on Sarah herself (which we'll suppose will be updated in response to various game events):

```
sarah : Person ...
    mood = mHappy
    ...
;
```

Then we could define various TopicGroups within, say, sarahFollowing:

```
+ sarahFollowing : AccompanyingState
    ...
;

++ TopicGroup
    isActive = (sarah.mood == mHappy)
;

+++ AskTopic @sarah
    "<q>How are you?</q> you enquire.\b
    <q>Oh - absolutely fine!</q> she assures you."
;

+++ AskTopic @goldenBanana
    ...
;

++ TopicGroup
    isActive = (sarahMood == mAngry)
;

+++ AskTopic @sarah
    "<q>How are you feeling?</q> you ask.\b
    <q>Absolutely livid!</q> she declares through gritted teeth."
;

+++ AskTopic @goldenBanana
    ...
;
```

Note that this doesn't stop you *also* locating TopicEntries directly in the actor state as well, if you want them to be common to all of Sarah's moods (say). Or you could put a TopicEntry both in an actor and in a TopicGroup, if, for

TADS 3 Tour Guide

example you wanted one response when Sarah was angry, and another when she was in any other mood. To ensure that the one in the 'angry' TopicGroup took precedence when Sarah was angry you could then set that TopicGroup's **matchScoreAdjustment** property to boost the score of all its TopicEntries (e.g., setting matchScoreAdjustment to 50 would effectively add 50 to the matchScore of all the TopicEntries located within it).

Another problem you could use TopicGroup to solve is the case where you want an actor to give the same response to a number of conversational commands in several but not all of its ActorStates. For example, suppose that than an actor progresses through actor states to which we'll give the unimaginative names state1, state2, state3 and state4 over the course of the game, and that there's a group of topics for which you want one answer if the actor's in state1 or state2 and another if it's in state3 or state4. Rather than having to duplicate the TopicEntries under both states, you could create a couple of TopicGroups directly in the actor:

```
bob : Person 'bob' 'Bob'
    isHim = true
    ...
;

+ TopicGroup
    isActive = (getActor.curState is in (state1, state2))
;

++ AskTopic @bob
    ...
;

++ AskTopic @dora
    ...
;

+ TopicGroup
    isActive = (getActor.curState is in (state3, state4))
;

++ AskTopic @bob
    ...
;

++ AskTopic @dora
    ...
;
```

Note that in this case, if any of these individual actor states have [DefaultTopics](#) that might mask bob and dora, these actor states would need to define `excludeMatch = [bob, dora]`.

Of course, these are just two suggested uses for TopicGroup; there are doubtless many more.

19.7.25. DefaultTopics

19.7.25.1. DefaultTopics - Overview

There is no way in which any game author can provide responses for every topic players will attempt to raise with NPCs via GIVE, SHOW, TELL and ASK commands during the course of a game. On the other hand seeing "Sarah does not respond" (which is what the library will display if the author does not provide any other response) as the response to a large variety of GIVE, SHOW, TELL and ASK commands will probably not create a terribly favourable impression on the player. The various types of DefaultTopic are provided to deal with this; rather than have "X does not respond" they allow the author to provide customised default handling for the various conversational commands when the player attempts to ASK, TELL, GIVE or SHOW something for which no specific response has been programmed.

Roughly speaking, there is a DefaultTopic type corresponding to each type of TopicEntry:

[DefaultAnyTopic](#)
[DefaultAskForTopic](#)
[DefaultAskTellTopic](#)
[DefaultAskTopic](#)

TADS 3 Tour Guide

[DefaultConsultTopic](#)
[DefaultGiveShowTopic](#)
[DefaultGiveTopic](#)
[DefaultInitiateTopic](#)
[DefaultShowTopic](#)
[DefaultTellTopic](#)

Note that there is also a [DefaultAnyTopic](#) that matches any conversational command for which the response is not specifically defined.

One thing to watch out for is that a DefaultTopic defined on an ActorState will mask any specific TopicEntries defined on the actor. So, for example, if we defined a DefaultAskTellTopic in sarahFollowing none of the AskTopics and TellTopics defined directly on sarah will be reachable when she's in the sarahFollowing state. This means that if you define any common TopicEntries on the actor, you'll need to define the corresponding DefaultTopics there as well. As of TADS 3.0.6p, however, there are two ways round this. The first, introduced in TADS 3.0.6n uses the **excludeMatch** property of the DefaultTopic, which we'll demonstrate with [DefaultGiveTopic](#). The second, new to TADS 3.0.6p, uses the **deferToEntry(other)** method. Note that in any case a DefaultTopic in an ActorState will only mask the corresponding type of TopicEntry in the actor, so that, for example, if you only use (say) AskTopics and TellTopics directly on the Actor, it would be safe to put DefaultGiveTopics, DefaultShowTopics and DefaultAskForTopics in the ActorStates. Thus, if you put any TopicEntries directly on the Actor, the corresponding type of DefaultTopic should go on the actor, not the ActorState. The converse does not necessarily apply: if you wish to handle all the DefaultTopics on the actor rather than making them state-specific there's absolutely no reason why you should not do so even if you define no other TopicEntries directly on the actor.

Apart from that *programming* DefaultTopics isn't particularly difficult; the difficulty lies in devising ones that work well. One might suppose, for example, that a suitable DefaultAskTopic could say something like "<q>I don't know anything about that,</q> she confesses, and a reasonable DefaultAskForTopic might be something like, "<q>I haven't got it,</q> she points out", but this could lead to a transcript such as the following:

>ask sarah about her home

"I don't know anything about that," she confesses.

>ask sarah for sex

"I haven't got it," she points out.

Such exchanges may be regarded as less than felicitous. The trick is to devise default responses that make some kind of sense no matter what they're responses to, and which at the same time help develop the character. Since the player is likely to be seeing a lot of default responses, it's also probably a good idea for DefaultTopics to be combined with a ShuffledText list to provide some variety to them. Clearly this is more important with a major NPC the player character will be interacting a lot with than with a minor one.

19.7.25.2. DefaultAskTopic

A DefaultAskTopic responds to any ASK X ABOUT Y command when no specific response (AskTopic or AskTellTopic) has been provided for Y.

The trick with a DefaultAskTopic is to provide a series of responses that won't jar too much no matter what's asked. That means it's no good having NPCs claiming they don't know the answer to the question, since the player could ask something they plainly do know about (e.g. ASK SARAH ABOUT HER MOTHER). Instead, one must have the NPC ignore the question, refuse to answer (graciously, politely, rudely, indignantly or however else you want the NPC portrayed) or give some completely neutral and uninformative reply (however wrapped up). Thus, for Sarah, we could define a DefaultAskTopic (on the sarah object) thus:

```
+ DefaultAskTopic, ShuffledEventList
[
  '<q>Can we talk about something else?</q> she suggests. ',
  '{The sarah/she} mutters something you don\'t quite catch. ',
  'In reply she merely cocks one eyebrow and gives an enigmatic smile. ',
  '<q>I\'m sorry,</q> she says, <q>my mind was elsewhere. Did you say something?</q> ',
  '<q>Let\'s discuss that some other time.</q> she replies. ',
  'Just at that moment, {the sarah/she} appears to be overcome by a
    fit of coughing. ',
  'She appears not to hear you. ',
  '<q>Well,</q> she says, proceeding to give a brief reply. '
]
```

;

Ideally, it would be good to extend the list, since the player is likely to see its members a good number of times, though it becomes difficult to think of many new ways of not saying anything. In this kind of case, nevertheless, you almost certainly want to use a `ShuffledEventList` to vary the responses.

19.7.25.3. DefaultTellTopic

A `DefaultTellTopic` responds to any TELL X ABOUT Y command when no specific response (`TellTopic` or `AskTellTopic`) has been provided for Y.

A `DefaultTellTopic` is relatively easy to devise, since it need comprise no more than a non-committal response to whatever the player character chooses to talk about. Here's how we might implement a `DefaultTellTopic` for Sarah (which we'll put directly in the `sarah` object, not one of her `ActorStates`).

```
+ DefaultTellTopic, ShuffledEventList
[
  '{The sarah/she} listens politely while you rattle on. ',
  '<q>How interesting!</q> she declares, <q>You must tell me more some time.</q>',
  '<q>Really!</q> she declares, <q>how fascinating!</q>',
  '{The sarah/she} does her best to suppress a yawn but can\'t quite
  manage it. ',
  '{The sarah/she} listens attentively, apparently hanging on your every
  word. ',
  '<q>Yes, I see,</q> she nods understandingly. ',
  '<q>Well, well!</q> she smiles. '
];
```

Of course this will not be entirely player-proof, in that the determined player can generate some fairly bizarre exchanges, such as

>tell sarah about the worst experience you've ever had
"Well, well!" she smiles.

But it will probably do well enough for most game-related exchanges.

19.7.25.4. DefaultAskTellTopic

A `DefaultAskTellTopic` effectively combines the functionality of a `DefaultAskTopic` and a `DefaultTellTopic`, i.e. it will handle any ASK ABOUT or TELL ABOUT command for which no specific `TopicEntry` has been provided.

It can be convenient to have a single `DefaultTopic` to handle both sets of commands, thus avoiding the need for a separate `DefaultAskTopic` and `DefaultTellTopic`, although it may be slightly harder to devise a series of bland responses that look equally convincing in response to both ASK ABOUT and TELL ABOUT. The `DefaultAskTellTopic` is perhaps best used in situations where the author does not expect it to be so heavily employed, perhaps on relatively minor NPCs, or in little-used `ActorStates`, or as a means of catching irrelevant responses in a [Conversation Node](#) (see below).

By way of illustration, we'll provide `DefaultAskTellTopics` for the curator and Solomon, which would be placed in their respective `InConversationStates`. We'll start with the curator:

```
++ DefaultAskTellTopic, ShuffledEventList
[
  '<q>Right,</q> {the curator/he} nods. ',
  '<q>Hm.</q> he replies. ',
  '<q>I never discuss that sort of thing when on duty.</q> he tells you. ',
  '{The curator/he} breaks into a long, boring and utterly incomprehensible
  explanation that you rapidly stop listening to. ',
  '{The curator/he} proceeds to tell you everything he knows about ' +
  gTopicText + ', until you wish you\'d never raised the subject. '
];
```

It should be pointed out, of course, that a list of default responses such as this will in any case seem less jarring if the

TADS 3 Tour Guide

author has anticipated the topics players are likely to raise and provided answers for them. In reality, we should, at the very least, have provided AskTopics to cover the exhibits in general, the specific exhibits on display, and the golden banana. The only reason for not doing so is that such further examples of AskTopics would serve no useful purpose in the course of this tutorial, so that, once again, they must be left as an exercise for the reader.

It is, of course, possible to write a DefaultAskTellTopic with a single response. Since King Solomon allegedly enjoyed an encyclopedic knowledge of every subject under the sun, one might almost get away with:

```
++ DefaultAskTellTopic
    "{The solomon/he} nods sagely and proceeds to treat you to the breadth
    and depth of his wisdom on the subject of <<gTopicText>>. "
;
```

The player may feel this default response becomes inappropriate if Solomon is asked about nuclear physics, George W. Bush, or human cloning, but a player who raises such topics in the course of a conversation with the ancient Israelite king can hardly expect very sensible responses. Again, this kind of default response will be a bit less jarring if we supply suitable AskTopics and TellTopics to cover things the player could reasonably be expected to try conversing with Solomon about.

19.7.25.5. DefaultGiveTopic

A DefaultGiveTopic responds to any GIVE X TO Y command when no specific response (GiveTopic or GiveShowTopic) has been provided for X. Note, however, that this does not mean that a DefaultGiveTopic will respond to any old GIVE X TO Y, no matter what the X. For the DefaultGiveTopic to be activated, X must be something that the Player Character *could* give to Y, i.e. it must be an object that the PC is either holding or able to pick up, i.e. a portable object in scope. This makes DefaultGiveTopics (and DefaultShowTopics) rather easier to write than DefaultAskTopics and the like, since with a DefaultGiveTopic there is a predefined list of things the player can actually try to give the NPC.

We'll give Sarah two DefaultGiveTopics, the first of which we'll put in the sarah object directly:

```
+ DefaultGiveTopic, ShuffledEventList
[
    '<q>No thank you,</q> {the sarah/she} declines after a cursory glance. ',
    '<q>I\'d hang on to that if I were you,</q> she says, <q>it\'s no use to me.</q>',
    '<q>How kind!</q> she declares, with an ironic twinkle in her eye, <q>But I
    think I\'ll let you keep it!</q>'
]
;
```

Now we'll add a second one to the sarahFollowing state, that provides a list of responses more appropriate to when Sarah is following the player character around:

```
++ DefaultGiveTopic, ShuffledEventList
[
    '<q>I\'ll let you carry that,</q> she replies with a pert smile. ',
    '<q>Why don\'t you keep it for now?</q> she suggests. ',
    '<q>Keep it - I\'m sure your need is greater than mine.</q> she insists. '
]
;
```

If you compile the game and try it now, you should eventually come up against a problem: we've now made it impossible to give Sarah a gas-mask once she's in the sarahFollowing state, since the GiveTopic for gas masks is directly in the sarah object, and a DefaultTopic in an ActorState masks any TopicEntries of the same kind located directly in the actor (so that a DefaultGiveTopic located in sarahFollowing will make all GiveTopics located directly in sarah unreachable when Sarah's in her sarahFollowing state). Fortunately, there is a way round this: you can use the **excludeMatch** property of any DefaultTopic to list the objects you *don't* want it to match, which allows TopicEntries from a wider scope to match those objects after all. In this case, we have a GiveTopic on sarah that matches the two gas masks in the game, so all we need to do is explicitly exclude them from the list of objects that this DefaultGiveTopic matches:

TADS 3 Tour Guide

```
++ DefaultGiveTopic, ShuffledEventList
[
  '<q>I\'ll let you carry that,</q> she replies with a pert smile. ',
  '<q>Why don\'t you keep it for now?</q> she suggests. ',
  '<q>Keep it - I\'m sure your need is greater than mine.</q> she insists. '
]
excludeMatch = [gasMask1, gasMask2]
;
```

If there were many more gas masks in the game, and you wanted might keep on adding objects of class GasMask, this method of specifying exclusions might become error prone. One way to automate the process might be to make the DefaultGiveTopic a PreinitObject that automatically builds the list of excluded objects from anything that's of class GasMask:

```
++ DefaultGiveTopic, ShuffledEventList, PreinitObject
[
  '<q>I\'ll let you carry that,</q> she replies with a pert smile. ',
  '<q>Why don\'t you keep it for now?</q> she suggests. ',
  '<q>Keep it - I\'m sure your need is greater than mine.</q> she insists. '
]
execute()
{
  local obj = firstObj(GasMask);
  while(obj != nil)
  {
    excludeMatch += obj;
    obj = nextObj(obj, GasMask);
  }
}
;
```

Though this is probably overkill in the present context. In the present context, the simplest method would be to override matchScore:

```
++ DefaultGiveTopic, ShuffledEventList, PreinitObject
[
  '<q>I\'ll let you carry that,</q> she replies with a pert smile. ',
  '<q>Why don\'t you keep it for now?</q> she suggests. ',
  '<q>Keep it - I\'m sure your need is greater than mine.</q> she insists. '
]
matchTopic(fromActor, topic)
{
  return topic.ofKind(GasMask) ? nil : matchScore;
}
;
```

But note that this is only safe if we *know* that it's only ever gas masks that we'll want this DefaultGiveTopic to skip. It might become error prone if, for example, we later wanted to Sarah also to accept the silverGizzmo in a GiveTopic on sarah, and just went ahead and put excludeMatch = [silverGizzmo] on the DefaultGiveTopic (whereas the PreinitObject method would be quite safe in such an instance, since it would simply add all objects of class GasMask to what was already in the DefaultGiveTopic list). A safer way of overriding matchTopic to do what we want would be:

```
matchTopic(fromActor, topic)
{
  return topic.ofKind(GasMask) ? nil : inherited(fromActor, topic);
}
```

Perhaps the most general solution overall is to modify DefaultTopic to exclude classes as well as individual objects, thus:

```
modify DefaultTopic
  excludeClass = []
  matchTopic(fromActor, topic)
  {
    return excludeClass.indexWhich({x: topic.ofKind(x)}) != nil
      ? nil : inherited(fromActor, topic);
  }
;
```

TADS 3 Tour Guide

Then our DefaultGiveTopic becomes simply:

```
++ DefaultGiveTopic, ShuffledEventList, PreinitObject
[
  '<q>I\'ll let you carry that,</q> she replies with a pert smile. ',
  '<q>Why don\'t you keep it for now?</q> she suggests. ',
  '<q>Keep it - I\'m sure your need is greater than mine.</q> she insists. '
]
excludeClass = [GasMask]
;
```

And we can quite safely keep adding to excludeClass and excludeMatch if we need to. Note, however, that this modification will only work as expected for DefaultTopics that match Things; if the DefaultTopic matches a ResolvedTopic, then topic will be ofKind ResolvedTopic, and the excludeClass list won't do much (although provided we don't specify excludeClass = [ResolvedTopic] this probably won't do too much harm).

An alternative approach would be to use the **deferToEntry(other)** method. This allows any TopicEntry to 'defer' to a TopicEntry in a lower priority TopicDatabase. In matching topics the priority is (1) the current ConvNode (if any), (2) the current Actor State and (3) the current Actor. We can therefore use deferToEntry to have a DefaultTopic on, say, an ActorState defer to a more specific TopicEntry on the actor; the method must return true if the deferral is to take place and nil otherwise. Most commonly, we might want a DefaultTopic on an ActorState to defer to a specific TopicEntry that's matched on the actor but not to a DefaultTopic that's defined there. This approach will work here, so instead of the previous definition we could define:

```
++ DefaultGiveTopic, ShuffledEventList
[
  '<q>I\'ll let you carry that,</q> she replies with a pert smile. ',
  '<q>Why don\'t you keep it for now?</q> she suggests. ',
  '<q>Keep it - I\'m sure your need is greater than mine.</q> she insists. '
]
deferToEntry(other) { return !other.ofKind(DefaultTopic); }
;
```

19.7.25.6. DefaultShowTopic

A DefaultShowTopic responds to any SHOW X TO Y command when no specific response (ShowTopic or GiveShowTopic) has been provided for X. Note that, as with the DefaultGiveTopic, X must be an object in scope, but, unlike the DefaultGiveTopic, it need not be portable. Thus, in Sarah's starting location, the DefaultShowTopic will be triggered by commands such as SHOW SARAH THE DOOR and SHOW SARAH THE SHIP (but not SHOW SARAH THE LAKE, since the lake is a Decoration object).

This means that we must either be careful not to supply any default responses that presuppose that the object shown to Sarah can be picked up and physically handed over, or else devise some method of supplying different responses for portable and non-portable items. Just for fun, we'll go down the second route, by overriding handleTopic to test whether the object matched is portable or not, and using the eventList property to list the portable responses and the topicResponse property to handle the non-portable cases.

Here's how we'll define the DefaultShowTopic for Sarah (again, put this in sarah, not one of her ActorStates):

```
+ DefaultShowTopic, ShuffledEventList
"{The sarah/she} glances at {the dobj/him} and remarks, <q>I'd say
it's {a dobj/he}, <<rand('definitely', 'palpably', 'undeniably')>> {a dobj/he}.
How <<rand('remarkable', 'quaint', 'nice', 'strange', 'odd')>>!</q> "

[
  '<q>Very nice, I\'m sure.</q> she remarks, with no obvious enthusiasm. ',
  '<q>How interesting!</q> she declares. ',
  'She examines {the dobj/him} and hands {it dobj/him} back to you. ',
  '<q>It\'s {a dobj/he}.</q> she observes, <q>So?</q> '
]
/* We override this method to distinguish between portable and other objects */
handleTopic(fromActor, obj)
{
  /* note the invocation */
  noteInvocation(fromActor);
```

TADS 3 Tour Guide

```
/* set pronoun antecedents if possible */
setTopicPronouns(fromActor, obj);

/* check to see if the matched object is portable */
if (!obj.ofKind(NonPortable))
{
    /* invoke our script */
    doScript();
}
else
{
    /* show our simple response string */
    topicResponse;
}
}
;
```

Note the use of the `dobj` parameter substitution strings to get at the object that Sarah has been shown. Since SHOW TO is a TIAction, any valid SHOW TO action will have a valid direct object, so we can use these parameter strings to describe it (or `gDobj` to get at the object itself). This would also work with GIVE TO (but not with ASK ABOUT, ASK FOR, or TELL ABOUT). Note also how we use the `rand()` function to provide some variety to the `topicResponse` without actually specifying a list of separate responses. When the `rand()` function is supplied with the list, it selects one of the elements at random.

19.7.25.7. DefaultGiveShowTopic

A `DefaultGiveShowTopic` combines the functionality of a `DefaultGiveTopic` and a `DefaultShowTopic` by responding to any GIVE X TO Y or SHOW X TO Y command for which a specific `GiveTopic`, `ShowTopic`, or `GiveShowTopic` has not been provided. The further remarks relating to `DefaultGiveTopic` and `DefaultShowTopic` also apply here; the `DefaultGiveShowTopic` will be triggered by SHOW X TO Y provided X is in scope, but GIVE X TO Y only if, in addition, X is portable and the player character can take it (or is already holding it).

With these factors in mind, we can define a `DefaultGiveShowTopic` that should work reasonably well for the curator (put it under the curator talking state). Once again we make liberal use of the `rand()` function to provide some measure of apparent variety to the curator's response:

```
++ DefaultGiveShowTopic
    "<q><<rand('I\'m sorry', 'I\'m afraid', '', 'I regret')>> I don't have time
    to <<rand ('be bothered with', 'look at','study', 'inspect')>> {that dobj/him}
    <<rand('right now', 'at the moment', 'today')>>.</q> he tells you."
;
```

We'll assume that Solomon, an avid collector of wisdom on all topics, is fascinated by anything he's shown (put this in the `solomonTalking` state):

```
++ DefaultGiveShowTopic, ShuffledEventList
[
    '<q>Fascinating!</q> {the solomon/he} declares. ',
    '<q>So that\'s {a dobj/he}, how interesting!</q> he remarks. ',
    '{The solomon/he} examines {the dobj/him} carefully, <q>This is
    how one\'s wisdom grows.</q> he tells you, <q>By careful
    observation of all things under the sun.</q> ',
    '<q>Thank you,</q> he says, looking at {it dobj/him} carefully,
    <q>The more I observe, the wiser I become.</q> '
]
```

TADS 3 Tour Guide

19.7.25.8. DefaultAskForTopic

A DefaultAskTopic responds to any ASK X FOR Y command when no specific response (AskForTopic or AskAboutForTopic) has been provided for Y.

A satisfactory DefaultAskForTopic is actually quite tricky to write, since there are so many possibilities. The player might ask for a game object, or a topic that's been defined, or for something not defined at all in the game world, and all would match. A satisfactory response to asking for a game object depends where the object is - held by the player character, or held by the NPC, or held by neither but in sight, or out of sight altogether. Any of these responses would need to be different from asking for something abstract like ADVICE, HELP or DIRECTIONS.

One approach might be to modify the DefaultAskForTopic class to handle the most obvious situations by overriding its handleTopic method. We can start by determining whether what the player has asked for is a game object, a topic, or something not defined in the game at all. If it's either a topic or something not defined we'll leave the DefaultAskForTopic to handle it in the normal way (either through topicResponse or an EventList), with the option to provide different handling if it's a defined topic. If, on the other hand, the player has asked for an object we'll try to provide default intelligent handling according to the location of the object, but we'll allow all the messages to be easily changed on individual DefaultAskForTopics:

```
modify DefaultAskForTopic
  handleTopic(fromActor, topic)
  {
    /* note the invocation */
    noteInvocation(fromActor);

    /* set pronoun antecedents if possible */
    setTopicPronouns(fromActor, topic);

    obj = topic.getBestMatch;
    if(obj == nil)
      inherited(fromActor, topic);
    else if(obj.ofKind(Thing))
      handleThing(fromActor);
    else if(obj.ofKind(Topic))
      topicMsg;
  }
  /* The object (if any) matched by this topic */
  obj = nil

  handleThing(fromActor)
  {
    if(obj.isIn(fromActor))
      alreadyHaveMsg;
    else if(obj.isIn(getActor))
      refuseMsg;
    else if(obj == getActor)
      askForOtherMsg;
    else if(obj == fromActor)
      askForSelfMsg;
    else if(getActor.canSee(obj))
      pointOutMsg;
    else
      dontHaveMsg;
  }

  /* The message to display if the player character asks for something he already has.
   If the player character is carrying the asked-for object in another container,
   the NPC points this out. */
  alreadyHaveMsg = "<q>You already have <<obj.theName>>, </q> <<getActor.theName>>
    points out<<obj.isDirectlyIn(gActor) ? '.' : ', <q>' + obj.itIsContraction +
    ' in that ' + obj.location.name + ' you\'re carrying.</q>'>> "

  /* The message to display if the requested actor has the object asks for but declines
   to hand it over */

  refuseMsg = "<q>No, I need <<obj.itObj>> myself.</q> <<getActor.itNom>> replies. "

  /* The message to display if neither the asker nor the askee has the object but
   the askee can see where it is */
```


TADS 3 Tour Guide

```
pointOutMsg = "<q>\^<<obj.itIsContraction>> just over there.</q> <<getActor.itNom>>
  observes, pointing at <<obj.location.ofKind(Room) ? 'the ground' :
  obj.location.theName>>. "

/* The message to display if neither the asker nor the askee has the object and the
  askee can't see it */
dontHaveMsg = "<q>I'm afraid I don't have <<obj.itObj>>,</q> <<getActor.itNom>>
  tells you. "

/* The message to display if the player asks for the NPC */
askForOtherMsg = "<q>That's me - here I am.</q> <<getActor.itNom>> tells you. "

/* The message to display if the player asks for himself/herself */
askForSelfMsg = "<q>You're right there.</q> <<getActor.itNom>> points out. "

/* By default we use the standard handling for a defined topic, but this
  can be overridden if desired. */
topicMsg()
{
  if(ofKind(Script)) doScript;
  else topicResponse;
}
;
```

Then, for Sarah, we could define an eventList for dealing with non-Thing objects, and override dontHaveMsg to provide a list of responses when she's asked for game objects she doesn't have:

```
+ DefaultAskForTopic, ShuffledEventList

[
  '<q>What do you need ' + gTopic.getTopicText + ' for?</q> she wonders. ',
  '{The sarah/she} shakes her head, <q>Sorry, I can\'t help you there,</q>
  she says. ',
  '<q>You must be joking!</q> she laughs. ',
  '<q>No, I think not.</q> she refuses with a firm shake of the head. '
]
dontHaveList : ShuffledEventList
{
  [
    '<q>I haven\'t got ' + lobj.aName + ',</q> she protests. ',
    '<q>I\'m afraid I don\'t have ' + lobj.itObj + '.</q> she tells you. ',
    '<q>What ' + gTopic.getTopicText + '?</q> she asks. ',
    '<q>What makes you think I\'ve got ' + lobj.itObj + '?</q> she demands. '
  ]
  lobj = (lexicalParent.obj)
}
dontHaveMsg { dontHaveList.doScript; }
;
```

In the above definition we use gTopic.getTopicText to get at whatever the player typed after ASK FOR. We define lobj = (lexicalParent.obj) on dontHaveList simply as a convenience (to avoid having to type lexicalParent.obj in all the strings we list).

We can now go ahead and provide DefaultAskForTopics for some of other NPCs. We'll start with Solomon, putting this one in the solomonTalking state:

```
++ DefaultAskForTopic, ShuffledEventList
[
  '<q>I\'m afraid I can\'t oblige you there.</q> he replies. ',
  '<q>I\'m renowned for my wisdom, not for ' + gTopic.getTopicText
  + ',</q> he points out. ',
  '<q>You\'ll have to go elsewhere for that.</q> he tells you. ',
  '<q>That\'s not in my power to give - I suggest you pray to the LORD our
  God, and see if he may graciously grant your request.</q> {the solomon/he}
  tells you. '
]
dontHaveMsg = "<q>Do I look as if I have such a thing about my person?</q>
  he demands. "
```

TADS 3 Tour Guide

;

Similarly, we can put a DefaultAskForTopic in the curatorTalking state:

```
++ DefaultAskForTopic, ShuffledEventList
[
  '<q>I never give ' + gTopic.getTopicText + ' when I\'m on duty.</q> he informs
  you pompously. ',
  '<q>It isn\'t my job to dispense ' + gTopic.getTopicText + '.</q> he tells you. ',
  '{The curator/he} mutters something inaudible under his breath. ',
  '<q>I should go elsewhere for that.</q> he advises you. '
]
refuseMsg = "<q>Now that I've got it, I intend to hang on to it.</q> he tells you. "
dontHaveMsg = "<q>It\'s not mine to give.</q> he tells you flatly. "
;
```

19.7.25.9. DefaultAnyTopic

A DefaultAnyTopic will respond to any conversational command at all (ASK, TELL, GIVE or SHOW) that is not otherwise provided for, subject to the scope restrictions on Give and Show that apply to DefaultGiveTopic, DefaultShowTopic, and DefaultGiveShowTopic noted above.

Since there is no way of knowing what the player typed to trigger a DefaultAnyTopic, this catch-all device must be used with caution. The situation where it's most useful is where an NPC is so intent on its own agenda that it's not really responding to what the player character (PC) has just said or done. This can occur in two rather different situations: the first is where the NPC is effectively ignoring the PC, or at least the specifics of what the PC is saying or doing (e.g. because the NPC is drunk or insane or preoccupied with some other activity, so its responses can't really be expected to connect with what the PC has just said); the second is where the NPC is pressing the PC for an answer and isn't going to be put off by an attempt to change the subject. The second of these possibilities generally occurs in Conversation Nodes, and we'll come to it then. The first might be illustrated by the demons NPC; being demons one might expect them to give some irrational and mischievous responses:

```
++ DefaultAnyTopic, ShuffledEventList
[
  '<q>Look at him! Look at him!</q> a pair of the demons squeal,
  <q>Shall we tear him? Shall we eat him?</q>',
  'The demons burst into hoots of demonic laughter. ',
  '<q>You stupid mortal!</q> one of demons screeches at you, <q>You
  understand nothing! Nothing at all!</q>',
  '<q>You don\'t thcare uth!</q> a lisping demon tells you, <q>Thtupid
  little human!</q>',
  '<q>Despair and die! Despair and die!</q> one of them cries at you,
  <q>I am Anthrapax, stealer of souls!</q>\b
  <q>And I am Princifax, leader of ghouls!</q> hisses a second.\b
  <q>And I, most fearsome of all, am Incometax, bleeder of fools!</q>
  declares a third.\b',
  '<q>Have you come here to die?</q> one of them asks, poking you in
  the ribs, <q>Death is good for humans; it get rid of that futility
  you call life. Just say the word and we\'ll relieve you of it. Won\'t
  we?</q>\b
  <q>Yes! yes!</q> they all cry (apart from the lisping demon who of course
  cries <q>Yeth! Yeth!</q>)',
  '<q>You don\'t like us much, do you?</q> sneers the ugliest of the demons,
  <q>Don\'t you know it\'s not politically correct to be prejudiced against
  demons?</q>\b
  <q>Not legal either.</q> one of his fellows agrees. <q>We demons have
  rights.</q>\b
  <q>So mind your manners or we\'ll sue you!</q> warns another, <q>Plenty
  of lawyers in hell, you know!</q>'
]

[
  'The demons howl at you derisively. ',
  'One of the demons gives you a sharp jab in the ribs. ',
  '<q>A human a day keeps the doctor away!</q> one of them chants, eyeing
  you hungrily. ',
  '<q>The only good human is a dead human.</q> Anthrapax opines. ',

```

TADS 3 Tour Guide

```
'<q>Get lost, foolish mortal!</q> another demon snarls. ',
'<q>You\'re powerless here, pathetic little mortal.</q> Princifax tells you. ',
'<q>He doesn\'t know when he\'s beaten!</q> one of the demons giggles from
the back of the pack. ',
'<q>Thtupid little human, thtupid little human!</q> says the lisping demon',
'<q>We hate <i>all</i> humans,</q> Incometax informs you, <q>But <i>especially</i>
you - you are the most pathetic and degraded specimen of a superfluous and
misbegotten species!</q> ',
'<q>Turn back, foolish mortal!</q> they chorus, <q>We won\'t be patient with
you much longer!</q> '
```

```
]
```

```
;
```

19.7.26. SuggestedTopics

19.7.26.1. SuggestedTopic

Although the player can ASK or TELL NPCs about anything, ASK THEM for anything, or attempt to GIVE or SHOW them anything that's available to be given or shown, only a comparatively few of these topics are likely to be implemented in your game. That may be fine if you want the players to have to guess what to ASK ABOUT etc., or because it should be reasonably obvious, but in some circumstances it may degenerate into a game of "guess the topic" (or even, "guess the wording of the topic"), which players may find simply frustrating.

To avoid this, you can provide players with a list of suggested topics, using subclasses of the SuggestedTopic class:

```
SuggestedAskTopic
SuggestedGiveTopic
SuggestedNoTopic
SuggestedShowTopic
SuggestedTellTopic
SuggestedTopicTree
SuggestedYesTopic
```

Note that there are no composite types of SuggestedTopic (such as SuggestedAskTellTopic), because we shall either Suggest asking or telling, but not both on the same topic (even if it's a composite kind of topic like an AskTellTopic). Which brings us to the point that SuggestedTopic is a *mix-in* class, which means that you add particular type of SuggestedTopic you want to the class list of the TopicEntry in question. For example, to make the TopicEntry for asking Sarah about her diamond ring a SuggestedAskTopic, we can change it to:

```
++ AskTopic, SuggestedAskTopic, StopEventList @diamondRing
[ '<q>What does your ring look like?</q> you ask.\b
  <q>It\'s a plain platinum band with a solitaire diamond.</q> she tells you. ',
  '<q>This ring - it\'s important to you?</q> you inquire.\b
  <q>Oh yes!</q> {the sarah/she} declares, <q>It\'s not it\'s monetary value, so
much; it\'s more a sentimental thing - you can\'t replace that with insurance
money. Besides, I was so <i>incensed</i> when this bloke snatched it off me -
why should he get away with it?</q>',
  new function {
    "<q>Have you any idea where your ring might be?</q> you ask.\b
    <q>Perhaps he dropped it back in there,</q> she suggests, nodding towards
    the open door,<q>let's go and see.</q><.p>";
    sarah.setCurState(sarahGuide1);
  },
  '<q>And the missing ring...</q> you begin.\b
  <q>... is a plain platinum band with a solitaire diamond.</q> she reminds you. '
]
name = 'the diamond ring'
timesToSuggest = 3
;
```

Then when player types TALK TO WOMAN to start a conversation, or uses the TOPICS command in the course of a conversation, the game will respond with "You could ask her about the diamond ring." In this case it will suggest it three times (because we have set timesToSuggest to 3) and then stop.

TADS 3 Tour Guide

The properties you may want to set on SuggestedTopic are:

- **name** - the name that will be given to this topic in the list of suggestions, after the phrase "You could ask about/ask for/tell about/show/give" You should *always* supply a value for this property, unless for some reason you override fullName.
- **fullName** - the title of the suggestion, suitable for following after "You could " (e.g. "You could ask about the diamond ring. "). You don't normally need to override this, since the library will construct a suitable phrase from the name property and the type of SuggestedTopic. If, however, you were creating your own custom kind of SuggestedTopic it might be useful to override this.
- **timesToSuggest** - rather than assuming that we have infinite curiosity about any particular topic, the game will assume we've exhausted our curiosity about it, that is it will stop suggesting it, after we have asked/told about (or shown or given) it this number of times. If you want this topic to go on being suggested *ad infinitum*, then set timesToSuggest to nil. By default, it's set to 1.

The default value of timesToSuggest *may* not be what you want where you have mixed in a Script class with a TopicEntry class to provide a list of responses. It may be that the subsequent responses are not significant and are just provided for variety in case the player should raise the topic again. Or it may be that they are sufficiently important that you want the player to work through them all. If the latter predominate among the SpecialTopics in your game, it may become just a little tedious to keep having to count the number of items in all the list and override timesToSuggest with the appropriate number. You may prefer it if the game automatically suggested a topic for the number of times there were fresh items in the list; in other words, instead of having timesToSuggest default to 1, you might prefer it to default to the number of different responses in the topic.

If you could prefer this, you can modify SuggestedTopic accordingly:

```
/* If the TopicEntry associated with a SuggestedTopic is also an EventList
 * keep on suggesting till we've been right through the list - i.e.
 * make timesToSuggest correspond to the length of the list.
 */

modify SuggestedTopic
  timesToSuggest = (ofKind(EventList) ? eventList.length() : 1)
;
```

Bear in mind, though, that this may still not always give the behaviour you want, and that you will probably still need or want to override timesToSuggest manually on at least some SuggestedTopics.

The occasions when the game will display the currently active SuggestedTopics are:

- When the player types a TALK TO, GREET or HELLO command
- When the player issue a TOPICS command
- Inside a [ConvNode](#) when there is an active [SpecialTopic](#)
- Under programmer control, following the use of a <.topics> tag in any output
- By a call to the NPC's **suggestTopics(explicit)** method (where explicit is either true or nil, as in a TOPICS or a TALK TO command respectively)

It's perfectly feasible to add a new SuggestedTopic class to match a new TopicEntry class. For example, a little while back we showed how to implement a custom [PraiseTopic](#). The only problem with it is that the player might never know it's there, or that PRAISE SARAH (for example) is a valid conversational command. The ideal solution to this is to create a SuggestedPraiseTopic that can be mixed in with PraiseTopic so that the player can discover its existence. Here's the definition we need:

```
SuggestedPraiseTopic : SuggestedTopic
  suggestionGroup = [suggestionPraiseGroup]
  fullName = 'praise {it targetActor/him}'
  name = '{it targetActor/him}'
;

suggestionPraiseGroup : SuggestionListGroup
  groupPrefix = "praise "
;
```

Add SuggestedPraiseTopic to the class list of all the PraiseTopics, and you now find 'praise her' among the list of conversational commands you can address to Sarah.

19.7.26.2. SuggestedAskTopic

A SuggestedAskTopic is the particular type of [SuggestedTopic](#) that prompts the player to Ask About such-and-such a thing.

For example, in addition to making the diamond ring a suggested topic for Sarah, we could do the same for asking about herself:

```
++ AskTopic, SuggestedAskTopic, StopEventList @sarah
[ {: "<q>What brings you here?</q> you wonder,\b
  <q>I was just taking a walk in the valley when some madman snatched my ring
  off me and ran in here, shouting at me to come and find it if I wanted it
  back.</q> she tells you, <q>So I followed him in. Now I
  just want to find my ring and get out of here.</q><<gSetKnown(diamondRing)>>" },
  {: "<q>And you are...?</q> you ask.\b
    <q><<sarah.makeProper>>,</q> she tells you, <q>My name's Sarah. Pleased
    to meet you - and I'll be even more pleased if you help me find my ring.</q>" },
    '<q>So you you\'ve never been in these caves before?</q> you ask.\b
    <q>No, never,</q> she replies, <q>and I wouldn\'t be here now if my ring
    hadn\'t been stolen. Caves aren\'t really my thing, though these
    aren\'t at all what I expected!</q> she nods towards the lake and the
    ship, and a pensive expression crosses her face, <q>I suppose it might
    be interesting to explore - but I really should get back.</q>',
    '<q>What are you so anxious to get back to?</q> you ask.\b
    <q>Fresh air and open sky, for a start!</q> she laughs. '
  ]
  name = 'herself'
;
;
```

(I'm assuming here that we've adopted my suggested modification to [SuggestedTopic](#), so that we don't need to set timesToSuggest by hand).

When the player character is conversing with Solomon, there's a couple of interesting objects he'll probably want to ask about anyway, but as we haven't provided any responses for them as yet, they'll furnish good examples of further SuggestedAskTopics. In both case they should be located inside the solomonTalking state. First the carbuncle:

```
++ AskTellTopic, SuggestedAskTopic, StopEventList @carbuncle
[
  '<q>That\'s a very fine looking carbuncle you have there.</q> you remark.\b
  <q>Indeed it is,</q> {the solomon/he} concurs, <q>I\'m told there\'s not
  another like it under the sun.</q>',

  '<q>How did you come by that carbuncle?</q> you ask.\b
  <q>It was one of the gifts the Queen of Sheba gave me
  when she came to be dazzled by my great wisdom.</q> he tells you.\b',

  '<q>That gem must be worth a fortune.</q> you observe.\b
  <q>No doubt.</q> the king concurs dryly. '
]
name = 'the carbuncle'
isActive = (!carbuncle.moved)
;
;
```

Note that we make this a SuggestedAskTopic although it's also an AskTellTopic; the player could TELL SOLOMON ABOUT CARBUNCLE as well as ASK SOLOMON ABOUT CARBUNCLE, but we only suggest one or the other (in this case ASK ABOUT). Note also that we provide an isActive condition that makes the topic (and hence the corresponding suggestion) active only until the carbuncle has been moved.

We should also add an entry to deal with the Baaras Root:

```
++ AskTopic, SuggestedAskTopic, StopEventList @baarasRoot
[
  {: "<q>What is that curious root you <<baarasRoot.moved ? 'had' :
    'have'>> there?</q> you inquire.\b
    <q>It grows in a ravine near a place called Baaras, hence we call
    it the Baaras-root - though the heathen call it Mandragora.</q> he
    tells you, <q>It has many interesting properties.</q>" },
    '<q>Tell me more about the properties of this Baaras-root,</q> you
```

TADS 3 Tour Guide

```
request.\b
<q>Well,</q> {the solomon/he} explains, <q>when growing, it emits
a bright light towards evening. At any time of day it proves hard
to pick, for it shrinks up to elude the grasp of anyone who approaches
with the intention of plucking it from the ground - although it can
be made to sit still by pouring on it, er, certain secretions of the
human body. Even then it is highly dangerous to obtain.</q>',
'<q>So why is the Baaras root so dangerous to pick?</q> you wonder.\b
<q>Why - because unless you manage to carry of the root, suspending it from
your hand, the mere touch proves fatal.</q> he tells you, <q>That\'s
why we normally use a dog - if you tie a dog to the root and walk
away, the dog will try to run after you, pulling the root from the
ground. It\'s instant death for the dog, of course, but it\'s
then perfectly safe to pick up the root and carry it away.</q>',
'<q>So what on earth makes the Baaras root worth all the bother
of plucking it up from the ground? Especially since it seems so
deadly!</q> you wonder.\b
<q>Ah. It\'s value is that it is also fatal to demons.</q> {the solomon/he}
tells you. <q>It\'s one of the most effective counters to demons I know.</q>',
'<q>So the Baaras root is good against demons?</q> you ask.\b
<q>Highly effective.</q> he assures you, <q>Nothing better.</q>'
]
name = 'the root'
;
```

In case you're wondering where I obtained these bizarre details, they come (more or less) straight from Josephus, *Jewish War*, 7.180-85. Although Josephus is there talking about first-century CE Palestine, he elsewhere notes (*Jewish Antiquities* 8.45-49) that God had granted Solomon particular expertise in the art of expelling demons, and that he, Josephus, had witnessed a fellow Jew successfully employing the incantations and the *root* prescribed by Solomon in an exorcism. It's a fair guess, then, that Josephus envisages Solomon prescribing the Baaras root for use against demons (which, of course, is not at all the same as supposing that Solomon actually ever did anything of the sort - but once again we are depicting the Solomon of legend here).

19.7.26.3. SuggestedTellTopic

A SuggestedAskTopic is the particular type of [SuggestedTopic](#) that prompts the player to Tell About such-and-such a thing.

For example, you might want to tell Sarah about the rockfall blocking the obvious way back out:

```
+ TellTopic, SuggestedTellTopic, StopEventList @tRockfall
[
  { : "<q>We won't get out the way we came it,</q> you warn her, <q>I'm
    afraid there's just been a rockfall, blocking the way back out
    to the outside world!</q> <<sarah.setKnowsAbout(tRockfall)>>\b
    <q>Oh wonderful!</q> {the sarah/she} declares, as if it was your
    fault. " },
  '<q>The way out back through the caves is blocked by a rockfall.</q>
    you say.\b
    <q>Yes, you already told me.</q> she reminds you. '
]
timesToSuggest = 1
name = 'the rockfall'
isActive = entranceTunnel.blocked
;

tRockfall : Topic 'rockfall' sarahKnowsAbout = nil;
```

Note that we set timesToSuggest here because we overrode it on [SuggestedTopic](#) to be the length of the list, but here only the first response is really worth seeing. In case at a later date we want any of Sarah's responses to depend on whether she knows about the rockfall or not, we call sarah.setKnowsAbout(tRockfall), although since Sarah is likely to be the only NPC interested in this we could perhaps more simply have used <.reveal rockfall>, which we could test with gRevealed('rockfall'). We also set isActive so that the player character can't talk about the rockfall before it happens. Note finally that we also need to define the tRockfall Topic; be careful not to mix this is with the TopicEntries, which might upset the location nesting tree.

19.7.26.4. SuggestedGiveTopic

A SuggestedAskTopic is the particular type of [SuggestedTopic](#) that suggests to the player that s/he could Give such-and-such to so-and-so.

For example, if we want to suggest to the player that he might give the carbuncle to the curator, we could add the following to the appropriate GiveTopic (previously defined):

```
++ GiveTopic, SuggestedGiveTopic @carbuncle
  topicResponse
  {
    "{The curator/he} takes the carbuncle and examines it carefully, then declares,
    <q>Wunderbar! Ausgezeichnet! This is the famous purple carbuncle of King Solomon,
    nicht wahr? And you are giving it to the museum? How kind, how very kind!</q>
    Pausing just to wipe the tears of excitement and gratitude out of his eyes, he
    continues, <q>I shall enroll you on our roll of honoured benefactors <i>at once</i>!
    Please, please, do feel free to inspect the special treasures in our benefactors'
    exhibition room any time you please!</q>";
    carbuncle.moveInto(getActor);
  }
  name = 'the carbuncle'
;
```

19.7.26.5. SuggestedShowTopic

A SuggestedAskTopic is the particular type of [SuggestedTopic](#) that prompts the player to SHOW such-and-such to so-and-so.

For example, if we want to suggest to the player that he might show the green ticket to the curator, we could add the following to the appropriate GiveTopic (previously defined):

```
++ GiveShowTopic, SuggestedShowTopic @museumTicket
  "<q>Thank you, that's fine.</q> {the curator/he} nods as he inspects your ticket,
  <q>Enjoy the exhibits!</q><.reveal ticket-shown>"
  name = 'the green ticket'
;
```

19.7.26.6. SuggestedYesTopic

A SuggestedYesTopic adds 'say yes' to the list of suggestions of what conversational commands the player can address to the currently conversing NPC. It would normally only be used in conjunction with a [YesTopic](#).

19.7.26.7. SuggestedNoTopic

A SuggestedNoTopic adds 'say no' to the list of suggestions of what conversational commands the player can address to the currently conversing NPC. It would normally only be used in conjunction with a [NoTopic](#).

19.7.26.8. SuggestedTopicTree

As we have seen, a [SuggestedTopic](#) class is included in the class list of a single [TopicEntry](#) to add that TopicEntry to the topic inventory, that is the list of topics that can be suggested to players. One limitation of a SuggestedTopic is that it applies only to the TopicEntry on which it is defined. This may sound like it's obviously the right behaviour, until one stops to consider what happens with [AltTopics](#). By default, if a TopicEntry which is also a SuggestedTopic has an active AltTopic, the suggestion will not be offered. This is because it is the AltTopic that it is now active. You can, of course, define the AltTopic to be a SuggestedTopic, and it will then be suggested. This is fine if you want the AltTopic and its parent TopicEntry to be treated as effectively separate topics for which curiosity is exhausted independently.

TADS 3 Tour Guide

For example, if you defined the following:

```
++ AskTopic, SuggestedAskTopic @bob
  "<q>What can you tell me about bob?</q> you ask.\b
  <q>Capital fellow!</q> he replies."
  name = 'Bob'
;

+++ AltTopic, SuggestedAskTopic
  <q>Have you heard about Bob?</q> you enquire.\b
  <q>Alas poor Bob, I knew him well!</q> he answers with a sad shake of the head."
  name = 'Bob'
  isActive = (bob.hasDied)
;
```

Then 'ask about Bob' will be offered once while Bob is still alive, and once more again when Bob passes away (assuming this causes bob.hasDied to become true). If this is what you want, then this is the way to do it. But you may instead want the TopicEntry and its AltTopics to be treated as a group that is to be suggested en bloc and exhausted en bloc. This would be the situation, for example, when the player needs only ask about the topic once, but the answer received will depend on the game state.

For example, consider the GiveShowTopics we defined for giving or showing Sarah the ring and the diamond while they are still separated, together with the [AltTopics](#) that make Sarah's response appropriate whichever order she's shown these objects in. In this case, if we wanted to make them SuggestedTopics, we'd also want to use SuggestedTopicTree:

```
++ GiveShowTopic, SuggestedTopicTree, SuggestedShowTopic @diamond
  "{The sarah/she} studies the gem carefully, <q>That certainly looks like it could
  be the diamond from my ring,</q> she decides, <q>But where's the ring?</q>
  <.reveal diamond-shown>"
  name = 'the diamond'
;

+++ AltTopic
  "<q>Yes, I think that's the diamond.</q> she nods eagerly, <q>Have you tried
  whether it fits the ring?</q>"
  isActive = gRevealed('ring-shown')
;

++ GiveShowTopic, SuggestedTopicTree, SuggestedShowTopic @ring
  "{The sarah/she} nods eagerly, <q>Yes, that's my ring!</q> she declares, but then
  her hand flies to her mouth, <q>But - oh my goodness - the diamond is missing!</q>
  <.reveal ring-shown>"
  name = 'the ring'
;

+++ AltTopic
  "<q>That's my ring, all right!</q> {the sarah/she} declares, <q>But you haven't
  fitted the diamond!</q>"
  isActive = gRevealed('diamond-shown')
;
```

Since this is a bit complicated, we'll add the explanation given as a comment in the library code:

A [SuggestedTopicTree is a] suggested topic that applies to an entire AltTopic group.

Normally, a suggestion is tied to an individual TopicEntry. This means that when a topic has several AltTopic alternatives, each AltTopic can be its own separate, independent suggestion. A particular alternative can be a suggestion or not, independently of the other alternatives for the same TopicEntry. Since each AltTopic is a separate suggestion, asking about one of the alternatives won't have any effect on the "curiosity" about the other alternatives - in other words, the other alternatives will be separately suggested when they become active.

In many cases, it's better for an entire set of alternatives to be treated as a single suggested topic. That is, we want to suggest the topic when ANY of the alternatives is active, and asking about any one of the alternatives will satisfy the PC's curiosity for ALL of the alternatives. This sort of arrangement is usually better for cases where the conditions that trigger the different alternatives aren't things that ought to make the PC think to ask the same question again.

Use this class by associating it with the root TopicEntry of the group of alternatives. You can do this most simply by

TADS 3 Tour Guide

mixing this class into the superclass list of the root TopicEntry:

```
+ AskTellTopic, SuggestedTopicTree, SuggestedAskTopic
// ... *. ;
++ AltTopic ... ;
++ AltTopic ... ;
```

This makes the entire group of AltTopics part of the same suggestion. Note that you must also include SuggestedAsk, SuggestedTellTopic, or one of the other specialized types among the superclass, to indicate which kind of suggestion this is.

19.7.26.9. limitSuggestions

One thing the topic inventory cannot know is which SuggestedTopics are actually reachable at any particular point. For example, suppose you have:

```
bob : Person 'bob', 'Bob'
    isHim = true
    ...
;

+ AskTopic, SuggestedAskTopic @mavis
  "<q>How's Mavis these days?</q> you ask.\b
  <q>Fine.</q> he replies, <q>Dead, of course, but otherwise fit and healthy. </q> "
  name = 'Mavis'
;

++ bobChatting : InConversationState
    ...
;

+++ DefaultAnyTopic
  "<q>I'd rather not discuss that right now.</q> Bob tells you. "
;
```

The problem here is that when Bob enters the bobChatting state, the AskTopic concerning Mavis is not actually reachable (since ASK BOB ABOUT MAVIS will be trapped by the DefaultAnyTopic, unless it's picked up by a more specific TopicEntry under the bobChatting state), but that "You could ask Bob about Mavis" will still be suggested as a possible topic of conversation. This is because the topic inventory mechanism has no means of knowing which TopicEntries are reachable and which are not, and so suggests everything that *might* be reachable.

The way round this is to use the limitSuggestions property. If limitSuggestions is set to true on an ActorState, then only the TopicEntries directly belonging to that ActorState will be suggested. In the above example we could add the line `limitSuggestions = true` to the bobChattingState, and the Mavis topic would not then be suggested when bob is in that state. Similarly, you can set limitSuggestions to true on a [ConvNode](#) to ensure that only the TopicEntries within that ConvNode are suggested (we'll be coming to ConvNodes next).

The full explanation of limitSuggestions given in the library source comments is as follows:

Flag: this database level should limit topic suggestions (for the TOPICS and TALK TO commands) to its own topics, excluding any topics inherited from the "broader" context. If this property is set to true, then we won't include suggestions from any lower level of the database hierarchy. If this property is nil, we'll also include any topic suggestions from the broader context.

Topic databases are arranged into a fixed hierarchy for an actor. At the top level is the current ConvNode object; at the next level is the ActorState; and at the bottom level is the Actor itself. So, if the ConvNode's limitSuggestions property is set to true, then the suggestions for the actor will include ONLY the ConvNode. If the ConvNode has the property set to nil, but the ActorState has it set to true, then we'll include the ConvNode and the ActorState suggestions.

By default, we set this to nil. This should usually be set to true for any ConvNode or ActorState where the NPC won't allow the player to stray from the subject. For example, if a ConvNode only accepts a YES or NO response to a question, then this property should probably be set to true in the ConvNode, since other suggested topics won't be accepted as conversation topics as long as the ConvNode is active.

19.8. Conversation Nodes

19.8.1. Conversation Nodes - Overview

The mechanisms we have seen so far, TopicEntries optionally combined with EventLists, allow some control over the how NPCs respond to conversational commands, but they don't really allow a conversation to develop in a structured fashion. It's true that with the use of AltTopics, the isActive property, <.reveal> tags and setKnowsAbout commands one can give *some* direction to a conversation, in that the game can keep track of what's been talked about and who knows what and adjust NPC responses accordingly, but this only gives a limited sense of progression to a conversation. One reason for this is that (with the partial exception of [InitiateTopic](#)) every mechanism we've looked at so far is a mechanism for programming NPC *response*; our NPC's can respond to ASK, TELL, GIVE and SHOW commands, but they don't seem to have any real way of asking questions of their own, let alone of starting an entire conversation on their own initiative.

This is where Conversation Nodes come in. With Conversation Nodes we can achieve the following:

- Have an NPC pose a question to the Player Character (and insist on an appropriate response)
- Have an NPC initiate a conversation with the Player Character
- Program a conversation that follows a definite thread (or branching threads), rather than merely have the NPC respond to a series of ASK and TELL commands.

The basic mechanism for this is the [ConvNode](#) class, which defines a Conversation node. When a Conversation node is active, the player can (optionally) be restricted to employing only the TopicEntries within that Conversation node (although, if you wish, you can also make the TopicEntries belonging to the current ActorState and the Actor available as well).

There are basically two ways to enter a Conversation Node. The simplest is via the use of a **<.convnode name>** tag in the text of an ordinary TopicEntry's response. For example, if I want my NPC to enter the banana-talk Conversation Node in response to ASK ABOUT BANANA I could define an AskTopic thus:

```
++ AskTopic @goldenBanana
  "<q>What you think about this banana thing?</q> you ask.\b
   <q>Banana?</q> he queries, <q>Do you mean <i>the</i> Banana - the Golden Banana of
Discord?</q>
   <.convnode banana-talk>"
;
```

The corresponding Conversation Node object could then be defined as simply as

```
+ ConvNode 'banana-talk';
```

With the appropriate TopicEntries located within the ConvNode. Note that a ConvNode must always be located either directly within the *Actor* to which it belongs, or (from TADS 3.0.9 onwards) within one of its ActorStates. Here we'll stick to putting ConvNodes directly within the Actor.

The second method of entering a Conversation Node is to call the **initiateConversation(ActorState, 'name')** method on the actor. For example, if I want an actor called fred to enter his fredChatting state and his 'fred-fuming' Conversation Node, I could call:

```
fred.initiateConversation(fredChatting, 'fred-fuming');
```

If I want to use this method to put an Actor into a Conversation Node *without* changing its ActorState, I simply call the method with nil as the first parameter, e.g.

```
fred.initiateConversation(nil, 'fred-fuming');
```

Alternatively, you can just call the setConvNode('name') method on the Actor:

```
fred.setConvNode('fred-fuming');
```

One way to *leave* a Conversation Node is to change to another Conversation Node using one of these methods.

TADS 3 Tour Guide

Another way is to do nothing, since, unless you arrange things to the contrary, the Conversation Node will only apply for one turn, and the game will leave the node after a single response. If you want the Actor to *stay* in the current Conversation Node (as you may), then you need to use a **<.convstay>** tag in the appropriate response or responses. For example, if the fuming Fred is forcing you to give a Yes or No answer to his irate question, you might put this in a DefaultAnyTopic that traps any other response other than Yes or No:

```
+ ConvNode 'fred-fuming';

...

++ DefaultAnyTopic
  "<q>Don't try to change the subject!</q> Fred snaps, <q>Have
  you stopped sleeping with my wife, yes or no?</q> <.convstay>"
;
```

Since this type of situation requires the player to be able to give a yes or no answer, the library defines a [YesTopic](#) and a [NoTopic](#) for the purpose (you can put a YesTopic or a NoTopic anywhere you can put any other kind of TopicEntry, but a Yes or No response will normally only make sense in the context of a Conversation Node).

This may seem a little unfair on the player, who may wish to deny sleeping with Fred's wife altogether. Well, we can handle this too, by means of a [SpecialTopic](#) (which can only be used within a ConvNode).

```
++ SpecialTopic 'deny sleeping with his wife' ['deny', 'sleeping', 'with', 'his', 'wife']
  "<q>I have never slept with your wife!</q> you reply, <q>Good grief, man, I've got
  better taste than that!</q>\b
  <q>So you're insulting her now, are you, you ungrateful wretch!</q>
  he cries, <q>Are you trying to tell me you find my wife so unattractive
  you don't <i>want</i> to sleep with her?</q><.convnode 'fred-wife-insulted'> "
;
```

If this doesn't seem entirely clear yet, don't worry. We'll go over it all step-by-step and in more detail as we explore how to use these feature in *The Quest of the Golden Banana*.

Note that as from TADS 3.0.10 the name we give to a Conversation Node (the quoted string name, that is, like 'fred-fuming' in the above example) only needs to be unique to the particular *actor*. We could give the name 'fred-fuming' to Conversation Nodes in two (or more) different actors, and the system would know which one we meant in any given context (namely the one that belongs to the actor who's being addressed).

19.8.2. ConvNode

ConvNode is the class used to set up a [ConversationNode](#) in an NPC's conversational responses. We'll discuss the principal methods and properties of the ConvNode class below, but first we'll clarify the concept by means of an example.

Once the player character (PC) gains access to the Benefactors' Room where the Golden Banana is stored, he'll find it enclosed in a glass case. To get at it, he'll need to cut the glass case open with a diamond. If Sarah is with him, she'll have the diamond in her ring, and he'll need to ask her for it (just as he did when he wanted to cut open the glass jar containing the [crystal](#)). This time, however, there's a catch; although Sarah can cut open the case, if she goes ahead and does so the microphone embedded in the pedestal under the case will pick up the sound, causing the curator to come in and discover the attempted theft. The solution to this will be for the PC to attach the sticking plaster (found in the first aid kit) over the microphone to prevent it from registering the sound. So if Sarah goes ahead and cuts open the glass case in response simply to being asked for her ring, the game will be lost in a way the player could not really have predicted. To prevent this from happening it would be better if Sarah offered to cut open the case and gave the player the choice whether she should go ahead or not. For this we need a ConvNode.

The first stage is to add a second AltTopic after the AskForTopic @diamondRing, an AltTopic that is triggered if Sarah is asked for the ring while she can see the glass case but the case is not yet open:

```
+++ AltTopic
  "<q>Can I borrow your diamond ring again?</q> you ask.\b
  <q>Don't tell me,</q> {the sarah/she} surmises, <q>you're
  thinking of cutting open that glass case - right?</q>
  <.convnode banana-case>"
  isActive = (getActor.canSee(bananaCase) && !bananaCase.isOpen)
;
```

TADS 3 Tour Guide

The important thing to note here is the use of `<.convnode banana-case>` to move Sarah into the Conversation Node after she's asked her question about cutting open the case. We now need to define the corresponding ConvNode (which must be located inside Sarah, *not* one of her ActorStates):

```
+ ConvNode 'banana-case'
  npcContinueList : ShuffledEventList
  {
    [ '{The sarah/she} looks at you expectently. ',
      '<q>Well, are we going to cut this thing open or aren\'t we?</q>'
      she asks. ',
      '<q>Did you want me to cut it open?</q> {the sarah/she} asks. ',
      '<q>You asked me for my ring,</q> {the sarah/she} reminds you,
      <q>Assuming you didn\'t just want to wear it, was it for some
      purpose, like trying to cut open that case?</q>'
    ]
  }
  limitSuggestions = true
;
```

The `npcContinueList` provides a list of messages that will be displayed if the PC fails to reply to Sarah's question at all but issues a non-conversational command (such as LOOK or EXAMINE BANANA). This reminds the player that Sarah has asked a question and is expecting a reply: it also makes her a bit more life-like, by having her show some human impatience for a reply. We set `limitSuggestion` to true to prevent a TOPICS command from listing any SuggestedTopics apart from the ones listed under the ConvNode (because we'll be blocking access to all other topics via a DefaultAnyTopic).

It's possible that the player may reply by trying changing the subject inconsequentially, e.g. by ASK SARAH ABOUT HERSELF or SHOW LAMP TO SARAH, which doesn't make for a very realistic conversation at this point. To trap this, we can use a DefaultAnyTopic which simply has Sarah repeat her question more insistently and then uses the `<.convstay>` tag to ensure that Sarah remains in this Conversation Node, still expecting a reply:

```
++ DefaultAnyTopic
  "<q>That glass case,</q> {the sarah/she} insists, <q>Should I try to cut it
  open, yes or no?</q><.convstay>"
;
```

This traps the invalid responses the player might make; we'll go on to give the valid ones when we come to look at [YesTopic](#), [NoTopic](#) and [SpecialTopic](#). But before we do that we need to look at the ConvNode class in a bit more detail. The principal properties and methods you may want to use on ConvNode are:

- **npcGreetingMsg** - use this to display a message when the NPC initiates a conversation (normally via a call to [initiateConversation\(\)](#)).
- **npcGreetingList** - use this as an alternative to `npcGreetingMsg` to provide a list of messages where the ConvNode may be initiated more than once.
- **npcContinueMsg** or **npcContinueList** - the [ActorState](#) class automatically displays the current ConvNode's continuation message (using either `npcContinueMsg` or `npcContinueList`, as appropriate) on each turn on which the ConvNode is active, and the player didn't address a conversational command to the NPC on the same turn.
- **noteLeaving()** - Note that we're leaving this conversation node. This doesn't do anything by default, but individual instances might find the notification useful for triggering side effects.
- **noteActive()** - Note that this ConvNode is becoming active. Our actor will call this method when the ConvNode is becoming active, as long as it wasn't already active. By default this schedules the topic inventory (to display the list of SuggestedTopics) if there are any SpecialTopics in the ConvNode (so if you want to insert any code here, you should remember to call `inherited`).
- **endConversation(actor, reason)** - Instances can override this for special behaviour on terminating a conversation (e.g. if the Player Character walks away while we're in the ConvNode we could have the NPC complain of his rudeness).
- **canEndConversation(actor, reason)** - lets a ConvNode prevent a conversation ending, by returning nil; in this and `endConversation` reason can be one of `endConvBye`, `endConvTravel`, or `endConvBoredom`. From TADS 3.0.10 `canEndConversation` can also prevent a conversation ending by returning the special value `blockEndConv`; this has the additional side effect that the caller will call `noteConvAction()` on the other actor, to prevent this actor from generating any further scripted remarks on the same turn. You should therefore return `blockEndConv` rather than nil if you display a message in `canEndConversation` that takes the form of the NPC explaining why s/he won't let you end the conversation, since if such a message has been displayed, you don't want to see one from `npcContinueMsg` or `npcContinueList` as well.
- **isSticky** - if true then the conversation remains in this node until a `<.convnode>` tag explicitly changes to another

TADS 3 Tour Guide

node (or no node at all). This can be useful when you are constructing a `ConvNode` where the NPC is insisting on an answer and won't give up till the PC gives one. By default `isSticky` is `nil`.

- **limitSuggestions** - if true then the only topics that will be suggested while the conversation is in this node are the topic entries that are actually within the node; otherwise (if `nil`, the default), a topic inventory request will list all the `SuggestedTopics` available from the actor's current `ActorState` as well. You'll normally want to set this property to true if you give the node one or more default topics that prevent topics from outside the node being reachable.

Note that the last two methods are only operative if the NPC's current `ActorState` is an `InConversationState`, since the notion of a conversation ending is only meaningful programmatically as an NPC switching out of an `InConversationState`. Thus, in the present example, where Sarah enters a `ConvNode` while in an `AccompanyingState`, these methods will never be called. We'll use an example where these methods are relevant [later](#). Basically, though, the three reasons a conversation might be terminated (when the NPC is in an `InConversationState`) are (a) if the player explicitly says goodbye (with a `BYE` command), corresponding to `endCondBye`; (b) if the player character walks away (the player types a movement command), corresponding to `endConvTravel`; or (c) if the player fails to issue a conversational command for long enough to exhaust the NPC's `attentionSpan` (`endConvBoredom`). The last two methods allow each of these situations to be handled differently, if required.

We'll see some more examples of `ConvNodes` when we come to look at the [initiateConversation](#) method.

19.8.3. YesTopic

A `YesTopic` simply responds to a `YES` command typed by the player. It is most useful when placed in a [ConvNode](#). For now we'll give a single example of a `YesTopic` we'll use to handle the player's replying `YES` to the question Sarah's just asked (about cutting open the glass display case). We'll also make it a [SuggestedYesTopic](#) so that it includes saying yes among the things suggested to the player at this point (for reasons that will shortly become apparent). Obviously, this `YesTopic` should be placed in the `ConvNode` we've just defined:

```
++ YesTopic, SuggestedYesTopic
  topicResponse
  {
    "<q>Yes, do you think you can manage it?</q> you ask.\b
    <q>Watch me!</q> she replies. ";
    nestedActorAction(sarah, CutWith, bananaCase, diamondRing);
    "<q>There you are!</q> she declares, <q>Easy!</q> ";
  }
;
```

19.8.4. NoTopic

A `NoTopic` responds to a `NO` command directed toward the currently conversing NPC. Like a `YesTopic` it is most useful inside a [ConvNode](#).

As an example, add the following immediately after the [YesTopic](#) we've just defined:

```
++ NoTopic, SuggestedNoTopic
  "<q>No, on second thoughts I think we'd better leave it for now.</q>
  you reply.\b
  <q>Very well.</q> she sighs. "
;
```

19.8.5. SpecialTopic

So far we have seen how we can allow the player to respond `YES` or `NO` to Sarah's question, but what if we want to respond with something else that doesn't fit into the `ASK/TELL/GIVE/SHOW` paradigm?

The answer is to use a `SpecialTopic`, which can respond to any string we care to define. Basically this works by providing the `SpecialTopic` with a `keywordList` containing a list of the words you want it to match. The special topic will match user input if the user input consists exclusively of words from this keyword list. The user input doesn't have to include all of the words defined here, but all of the words in the user's input have to appear here to match. So for example, if we defined:

TADS 3 Tour Guide

```
keywordList = ['lie', 'through', 'thru', 'your', 'teeth']
```

The SpecialTopic would match LIE THROUGH YOUR TEETH or LIE THRU TEETH or simply LIE.

Note that you can have more than one SpecialTopic in the same ConvNode that shares some of the same vocabulary, e.g.:

```
keywordList = ['gnash', 'your', 'teeth']
```

And both topics would work fine provided the player types enough words to determine which one is meant (e.g. LIE TEETH or GNASH TEETH), but if the only words typed are common to both SpecialTopics (in this case a command consisting of just TEETH) neither SpecialTopic will be matched and the game will treat what the player typed as a standard command. So if you do devise multiple SpecialTopics under the same ConvNode you need to treat overlapping wording with some care. In the example just given it would be rather perverse for the player to type simply TEETH, so you'd probably be okay; what you do need to avoid is any ambiguity in a SpecialTopic response a player might reasonably type.

Of course players can't be expected to *guess* the special phrasing you've devised for your SpecialTopics, so the library automatically treats every SpecialTopic as a SuggestedTopic and displays its name property as a prompt to the player, following the phrase "You could ". For example, for the two cases above you might define:

```
name = 'lie through your teeth'
```

and

```
name = 'gnash your teeth'
```

Then when the ConvNode is entered, the player will be prompted with:

"(You could lie through your teeth, or gnash your teeth.)

It follows that you want the name property to contain something that makes sense after "You could "

The SpecialTopic template simplifies these definitions by allowing you to define a SpecialTopic as:

```
SpecialTopic 'name' [keywordList]
    "topicReponse"
;
```

So you could define these two SpecialTopics thus:

```
++ SpecialTopic 'lie through your teeth' ['lie', 'though', 'thru', 'your', 'teeth']
    "<q>I've never even set eyes on Mabel!</q> you declare.\b
    <q>I don't believe you!</q> he snarls.
;

++ SpecialTopic 'gnash your teeth' ['gnash', 'your', 'teeth']
    "You gnash your teeth in frustration at his wild accusations
    while he glowers at you in mounting fury.\b
    <q>Answer my question, damn you!</q> he cries <.convstay> "
;
```

But since these aren't relevant to our game, we'll define one that is. Suppose that in addition to replying 'yes' or 'no' to Sarah's question about cutting open the glass case, we want to allow the player to ask her what she thinks. In this case we *could* define an adviceTopic: Topic with vocabulary such as 'advice/opinion' and then use an AskForTopic @adviceTopic. And in this case it might even be a simpler way of doing it! But to show how SpecialTopic works we'll do it the hard way with a SpecialTopic (which should be put in the same ConvNode as the NoTopic and YesTopic we've just defined; incidentally it's because a SpecialTopic always acts like a SuggestedTopic that we have made the NoTopic and YesTopic SuggestedTopics as well, so that the player can see that saying YES or NO are also valid options at this point):

```
++ SpecialTopic, StopEventList 'ask what she thinks'
    ['ask', 'her', 'sarah', 'what', 'she', 'thinks', 'do', 'you', 'think']
    [
        '<q>What do you think?</q> you ask.\b
        {The sarah/she} walks over to the case, raises her hand as if to
        start cutting it with her ring, but then pauses, peering down at
```


TADS 3 Tour Guide

```
    the plinth and
    running her hand along its carved decorative slat. She turns to
    you with a worried frown and says, <q>I think something\'s wrong.
    Are you sure you want me to go ahead with this?</q><.convstay> ',
    <q>What do you think is wrong?</q> you wonder,\b
    <q>I\'m not sure,</q> she admits, <q>Just <i>something</i>
    about this case. But maybe I\'m just being silly.
    So, do you want me to go ahead and cut it open?</q><.convstay> ',
    <q>What is it about the case that\'s worrying you?</q> you ask.\b
    <q>I can\'t quite put my finger on it,</q> she frowns, absently
    running her hand along the decorative slat round the polished oak
    plinth, <q>So shall I just go ahead and cut it open anyway?</q>
    <.convstay>'
]
;

+++ AltTopic, StopEventList
[
    <q>Do you think we should?</q> you ask.\b
    <q>Yes, let\'s go for it!</q> she enthuses,
    <q>So, shall I?</q><.convstay>',
    <q>Do you really think we should?</q> you ask.\b
    <q>Yes, I just said so!</q> she reminds you,
    <q>So what do you say?</q><.convstay>'
]
isActive = (microphone.attachedObjects.indexOf(stickingPlaster))
;
```

Note that although players will see the prompt "(You could ask what she thinks.)" the chances are that at least half of them won't suppose they have to follow the exact wording of the prompt, so we have to make sure our list of words in our keywordList covers all the things they're likely to type, including ASK HER WHAT SHE THINKS, ASK SARAH WHAT SHE THINKS and WHAT DO YOU THINK? Hence the list of keywords we supply here (though we don't need to worry about players ending the command with, say, a question-mark, since that's taken care of automatically).

The idea of the example above is that if the player does ask Sarah what she thinks, she gives a fairly good hint whether it's safe to proceed, and even a bit of a hint why it isn't if it isn't. The test in the topicResponse() method is for whether the sticking plaster is attached to the microphone. The idea is that, if it is, it will stop the microphone picking up the sound of the glass being cut, which will otherwise sound the alarm and bring the curator running in to investigate, which is why we set the objects up the way we did when [we first created the Benefactors' Room](#).

One point that may occur to you (or that you may in time discover) about SpecialTopics is that at least some players will be tempted to use them in inappropriate ways. That is, having once seen a prompt "ask what she thinks" as a possible response to a question, they may try to use it in other contexts as well, when it is no longer valid. Prior to version 3.0.8 TADS's response to this was potentially quite misleading, along the lines of "The word 'thinks' is not necessary in this game" or "The story doesn't understand that command." TADS 3.0.8 improves on this considerably by displaying the rather more accurate and informative message "That command can't be used right now." In order to do this the parser keeps track of a number of the most recent SpecialTopics command phrases displayed, and shows the "That command can't be used right now" if the inappropriate command matches one of these. The number of such SpecialTopic command phrases to be retained is held in the **maxEntries** property of the **specialTopicHistory** object. By default this number is set to 20, which will normally suffice for most games (since few players will remember more than the last 20 SpecialTopic command phrases displayed, while the number is not so large as to make a significant impact on performance). But if you like, you can set it to some other value; if you set it to nil, instead of checking against the most recent list the parser will scan *all* the SpecialTopics in the game (regardless of whether they've yet been suggested or not).

19.8.6. initiateConversation

While we can use the <.convnode> tag to have an NPC move into a [Conversation Node](#) in the middle of a conversation, and perhaps pose a question to the Player Character (PC) at that point, you might want a truly proactive NPC actually to start a conversation. To achieve that, you can call the **NPC's initiateConversation(ActorState, node)** method, where ActorState is the ActorState you want the NPC to change to (normally an [InConversationState](#)), or nil if you don't want a change of ActorState, and node is the Conversation Node that you want to become active. Note that the node parameter can either be the ConvNode's object name, or the quoted string used at the ConvNode's tag. We'll illustrate both below:

TADS 3 Tour Guide

We'll start with the ghost; we'll handle his entire appearance through ConvNodes, and we'll get him started with a call to initiateTopic(). First change the definition of graveyard.roomDaemon so it now reads:

```
roomDaemon
{
    inherited;
    if(!ghost.moved && !statue.isPulled)
    {
        ghost.moveInto(self);
        ghost.initiateConversation(ghostTalking, ghostNode);
        ghostAppearingEvent.triggerEvent(ghost);
    }
}
```

Now we may define the two ConvNodes for the ghost, together with their associated topics. Note how we have a very simple threaded conversation, that progresses from the first node to the second once the player accedes to the ghost's request (not that he has any real choice in the matter).

```
+ ghostNode : ConvNode 'ghostQuestion'
npcGreetingMsg = "<.p>A pale ghost rises slowly from one of the tombs,
then turns to you, pointing its ghostly finger straight at you.
\b<q>You!</q> the ghost cries,
<q>Yes you - the disturber of my statue! You are the one who must
carry out the sacred task! You are the one who must retrieve the
Golden Banana of Discord and cast it into the fires of Mount Gloom
before it falls into the hands of the Cabal! Will you carry out
this sacred quest? Will you?</q> <<gSetKnown(goldenBanana)>>
<<gPlayerChar.setHasSeen(ghost)>>"
canEndConversation(actor, reason)
{
    switch(reason)
    {
        case endConvTravel:
            "You are rooted to the spot. ";
            return nil;
        case endConvBye:
            "<q>Oh no, you don't get rid of me that easily!</q>
            cries the ghost. ";
            return nil;
        default:
            return nil;
    }
}
npcContinueMsg = "The ghost lets out a low moan and whispers
<q>My answer! My answer! I will have my answer! Do you
accept your quest?</q> "
;

++ NoTopic, SuggestedNoTopic
"<q>Er, no thanks. I'm off sacred quests at the moment...</q>
you begin feebly.\b
<q>No is not the right answer!</q> the ghost interrupts you,
<q>By toppling my statue you have shown you are the one!
So - will you accept your responsibilities and take up the
quest?</q><.convstay>"
;

++ YesTopic, SuggestedYesTopic
"<q>Well, if you insist...</q> you begin.\b
<q>I <i>do</i> insist!</q> roars the ghost, <q>So,
you have accepted the quest. Do you have any
questions about it?</q><.convnode quest-questions> "
;

++ AskTopic, SuggestedAskTopic @goldenBanana
"<q>What is this Golden Banana you're on about, anyway?</q>
you ask.\b
<q>It is that which you must find and destroy in your
quest!</q> the ghost answers, <q>Now, does this mean that
you are accepting your sacred task?</q><.convstay>"
name = 'the golden banana'
;
```


TADS 3 Tour Guide

```
++ DefaultAnyTopic
  "<q>Trying to change the subject will avail you nothing!</q>
  the ghost tells you, <q>When you've been dead as long as I have
  you become <i>very</i> single minded - almost monomaniac -
  so I <i>will</i> have my answer and you <i>shall</i> not leave
  this spot until I do. So, I ask again, do you accept the Sacred
  Quest of the Golden Banana?</q><.convstay>"
;

+ ConvNode 'quest-questions'
  npcContinueMsg = "<q>Do you have any further questions?</q> {the ghost/he}
  asks patiently. "
  canEndConversation(actor, reason)
  {
    if(reason==endConvTravel)
    {
      "You are too absorbed with {the ghost/him} to leave right now. ";
      return nil;
    }
    return true;
  }
  endConversation(actor, reason)
  {
    "<q>Well, goodbye then.</q> you say.\b
    <q>Farewell, Banana-Quester!</q> {the ghost/he} replies,
    <q>if you are sure you have no further questions?</q>\b";
    nestedAction(No);
  }
;

++ NoTopic, SuggestedNoTopic
  topicResponse
  {
    "<q>No, I don't think so.</q> you reply.\b
    <q>Very well, </q> he sighs, <q>Then I may depart, my job is done.
    Should you need to know more, consult the Great History in the Old
    Library. Nunc dimittis servum tuum Domine... </q>
    With a final moan, {the ghost/he} fades out of sight. ";
    ghost.moveInto(nil);
  }
;

++ YesTopic, SuggestedYesTopic
  "<q>Yes, I do.</q> you reply.\b
  <q>Would you care to be more specific?</q> {the ghost/he} suggests.
  <.convstay>"
;

++ AskTopic, SuggestedAskTopic, StopEventList @ghost
[
  {: "<q>Yes, who are you?</q> you demand.\b
  <q>I am - or was - <<ghost.makeProper()>> the Banana-Bearer.</q> he
  tells you. <q>I it was who countless aeons ago - or last week - ghosts
  don't have such a good sense of time, you see, time doesn't mean much
  when you're dead. Ah, where was I? Oh yes, I it was who countless
  ages ago chanced life and limb and sanity itself to snatch the
  Golden Banana of Discord from the slopes of Mount Gloom, and smuggle
  it past the dread demons of Hell-Fire Cavern into the world of men!
  A great hero men thought me! A great fool I was. Oh woe! Woe! Woe!</q>
  His moaning complete, the ghost enquires, <q>Right, any more questions?</q>
  <.convstay>" },
  '<q>But who are you?</q> you ask.\b
  <q>I told you I am - or was - Benedict the Banana-Bearer.</q> he
  reminds you. <q>Anything else?</q> <.convstay>'
]
  name = 'himself'
;

++ AskTopic, SuggestedAskTopic, StopEventList @goldenBanana
[
  '<q>So what exactly is this Golden Banana thingy? And why\'s
  it so important?</q> you enquire.\b
  '
]
```

TADS 3 Tour Guide

```
<q>The Golden Banana of Discord is the most Awesome Artifact in
the Universe!</q> cries {the ghost/he}, <q>When I fetched it from
the slopes of Mount Gloom I thought I was conferring immeasurable
benefits on my fellow man - health, wealth, happiness, power -
everything. Fool that I was! Oh yes, the Golden Banana is powerful
all right. Too powerful! And it is a terrible power! It is best
you do not know too much about it, lest you be tempted to imitate
the folly of others. But know this, should the Golden Banana of
Discord fall into the wrong hands, then all is lost, all! It will
be the end of life, death and Interactive Fiction as we know it!
Horrible! Terrible! So I implore you, find it and destroy it as
soon as you can - the only way it can be destroyed, by being cast
into the fires of Mount Gloom! So, is there anything else you
would know?</q> <.convstay>',
'<q>Is there anything else you can tell me about this Golden
Banana?</q> you enquire.\b
<q>I have told you all you need to know.</q> {the ghost/he}
assures you. <q>It is a terrible, powerful artifact that must
be destroyed. For you to know more is too dangerous. Any other
questions?</q><.convstay>'
]
name = 'the Golden Banana of Discord'
;

++ AskTopic, SuggestedAskTopic @tCabal
"<q>Who or what is this cabal you're so worried about?</q> you wonder.\b
{The ghost/he} looks distinctly shifty as he replies, <q>That's the
problem, no one knows!</q> he tells you, <q>That's why they're so
terrifying - they could be anyone, anyone at all! But whoever they
are, you <i>musn't</i> let them get the Golden Banana - or else -
or else we're all doomed - doomed!</q> He blinks a few more times,
then asks, <q>Was there anything else you wanted to know?</q><.convstay>"
name = 'the Cabal'
;

++ DefaultAnyTopic
"<q>Such things no longer concern me,</q> {the ghost/he} sighs,
<q>I am dead, after all. Now, was there anything else?</q><.convstay> "
;

tCabal : Topic 'cabal' ;
```

Note the use of the `endConversation` method in the second `ConvNode` to call a nested `No` command in the event of the player typing `BYE`; this way the code in the `NoTopic` is run whether the player leaves the `ConvNode` by typing `BYE` or `NO`. Note also that in order to make it work properly we have to give the first `ConvNode` a name property ('ghostQuestion') even though we entered it by referring to its object name (`ghostNode`).

If you try this out you'll find there's a small problem: if Sarah is accompanying the player character when the two of them encounter the ghost, an `initiateTopic` is triggered on Sarah, so *she* becomes the current interlocutor, with the result that if the player directs a response to the ghost or tries to a `TOPICS` command, it'll be taken as directed at Sarah, which isn't what we want before. There may be more than one way round this, but the way we'll use is to modify Sarah's `initiateTopic` so that it deflects conversation back to the ghost, like this:

```
+ InitiateTopic @ghostAppearingEvent
topicResponse
{
  "\b<q>Eek!</q> cries {the sarah/she}, clutching at your arm.
  <<getActor.setHasSeen(ghost)>><.p>";
  nestedAction(AskAbout, ghost, sarah);
}
;
```

Now we need to add an appropriate `AskTopic` in the `ghostNode ConvNode` to field this question:

```
++ AskTopic @sarah
"<q>Never mind her,</q> the ghost insists, <q>just answer my
question.</q><.convstay><.topics><.p>"
isActive = (gPlayerChar.canSee(sarah))
;
```

This will work fine after Sarah clutching the player character's arm, but will also be fine as a response to `ASK GHOST`

TADS 3 Tour Guide

ABOUT SARAH.

Well, that's quite enough about the ghost. Let's move on to the king. When the player character suddenly walks in on King Solomon, appearing as if by magic out of his hitherto empty bedroom, it's likely to give the monarch a bit of a shock, so you would imagine that Solomon won't wait to be addressed, but will ask who the intruder is. Again, we can use `initiateConversation` to handle this:

```
++ solomonExamining : ConversationReadyState
specialDesc {inherited; stateDesc; }
stateDesc = "He's staring at the table, deep in thought. "
isInitState = true
afterTravel(traveler, connector)
{
    if(traveler==gPlayerChar && !getActor.isProperName)
        getActor.initiateConversation(solomonTalking, 'solomon-surprised');
}
;
```

Then we then add a couple of `ConvNodes` (located in `solomon`) to get the conversation started:

```
+ ConvNode 'solomon-surprised'
npcGreetingMsg = "{The solomon/he} looks up, startled at your
appearance, and then says something that sounds like:
<q>Shlomo ani, melek yisrael, ha-hakhim ha-gadhol.</q>
At that point your ears mysteriously tune into the ancient tongue and
you realize that what he just said was, <q>I am <<solomon.makeProper()>>,
King of Israel, the great sage.</q> He continues, <q>Who are you,
where are you from, how did you get here? I know my bedroom was empty
just now, and no one could have got in, so who are you?</q> "
npcContinueMsg = "<q>I'm still waiting for your reply.</q> {the solomon/he}
reminds you. <q>Who are you? Why are you here?</q> "
canEndConversation(actor, reason)
{
    return dontEndSolomonConversation(reason);
}
limitSuggestions = true
;

++ SpecialTopic 'explain that you are from the future'
['explain', 'that', 'you', 'you\'re', 'from', 'the', 'future']
"<q>Actually, I'm a time-traveller, from about three thousand years
in your future.</q> you tell him.\b
<q>You travel in time?</q> he queries, <q>Then you must be some kind
of angel, a messenger of the Lord? So what message do you bear?
Speak!</q><.convnode solomon-angel>"
;

++ SpecialTopic 'lie' ['lie']
"<q>Well, actually I'm an angel in human form.</q> you lie.\b
The king nods sagely, <q>Well, you certainly look like no man
I have ever seen.</q> he concurs, <q>So perhaps an angel is what you
must be. What message do you bring?</q><.convnode solomon-angel>"
;

++ DefaultAnyTopic, StopEventList
[
    '<q>I think we should establish who you are before we attend to
anything else.</q> the king insists, <q>After all, it\'s not every day
that strangers suddenly emerge from my bedroom. So, what account do
you have to give of yourself?</q><.convstay>',
    '<q>First tell me who you are.</q> {the solomon/he} insists.<.convstay> '
]
;

+ ConvNode 'solomon-angel'
limitSuggestions = true
canEndConversation(actor, reason)
{
    return dontEndSolomonConversation(reason);
}
;
```

TADS 3 Tour Guide

```
++ SpecialTopic 'warn him about his son Rehoboam'
  ['warn', 'king', 'solomon', 'him', 'about', 'his', 'son', 'rehoboam']
  "<q>I've come to warn you about your son Rehoboam.</q> you improvise,
  <q>If you don't keep the lad in check, he'll split your kingdom after
  you've gone! Er, Thus says the Lord, because Rehoboam will chastise
  my people with scorpions, I shall tear them out of his hands, but I
  shall leave two tribes only for the sake of my servant David.</q>\b
  <q>You have been sent to warn me about my son?</q> Solomon muses,
  <q>Then I must watch him. It is true the company he keeps is not
  everything I could wish.</q><<hramNode.npcGreetingMsg>><.convnode hram>"
;

++ SpecialTopic 'condemn his foreign wives'
  ['condemn', 'his', 'foreign', 'wives']
  "<q>Er - Thus says the Lord, because you have not been content with a
  few wives, but have taken many unto yourself, and because you have allowed
  yourself to be led after the foreign gods that are no gods that these
  foreign women worship, I shall turn my face from you, and the kingdom
  shall be torn in two!</q> you improvise.\b
  <q>My wives!</q> the King tries to sound indignant, but looks more than
  a little guilty.<<hramNode.npcGreetingMsg>> <.convnode hram>"
;

++ SpecialTopic 'complain about his labour policy'
  ['complain', 'about', 'his', 'labour', 'policy']
  "<q>You've been using too much forced labour on your building projects.</q>
  you tell him, <q>The Lord is not pleased with this social injustice!</q>\b
  <q>Social injustice?</q> Solomon echoes, <q>What kind of language is that?
  Besides, I am building a Temple to the Lord, and the people are happy to
  participate in this great work!</q><<hramNode.npcGreetingMsg>>
  <.convnode hram>"
;

+ hramNode : ConvNode 'hram'
  npcGreetingMsg = "<.p>Solomon frowns at you thoughtfully for a moment,
  then continues slowly, <q>Angel or no angel, perhaps you have been sent
  to me to solve my problem with Hiram.</q> "
  npcContinueMsg = "<q>Yes, Hiram,</q> Solomon nods, <q>you must be here
  about Hiram.</q>"
  limitSuggestions = true
  canEndConversation(actor, reason)
  {
    return dontEndSolomonConversation(reason);
  }
;

++ DefaultAnyTopic
  "<q>Your majesty...</q> you begin.\b
  <q>If you can solve my problem, I'll reward you whether
  you're angel or no.</q> Solomon continues, evidently totally intent
  on this topic. <q>My neighbour King Hiram of Tyre has donated
  a fine bronze vessel for the temple. Unfortunately it has gone missing.
  This is very unfortunate; Hiram was displeased with the Galilean cities
  I gave him in payment for his contributions to my work on the Temple,
  and he is coming to discuss the matter. If he finds that I have mislaid the
  bronze bowl he specifically sent me to be dedicated on his behalf, he
  will <i>not</i> be pleased - and I cannot afford to upset him. So,
  you must recover this vessel for me.</q><<gSetKnown(bronzeBowl)>>"
;

dontEndSolomonConversation(reason)
{
  switch(reason)
  {
    case endConvTravel:
    case endConvBye:
      "Even you realize that you don't just walk out on a king. ";
      return nil;
    default:
      return nil;
  }
}
```

TADS 3 Tour Guide

Note how we've defined the `dontEndSolomonConversation` function to avoid having to repeat the same code in three `ConvNodes`. In both the above examples we've really only offered the player the illusion of choice: the path through the `ConvNodes` is the same whichever response the player chooses. It would, however, be possible to devise a branching conversation by having the NPC switch to different `ConvNodes` depending on the topic chosen. Note also the calls to `<<hiramNode.npcGreetingMsg>>` in the various `SpecialTopics` in the previous `ConvNode`; after some experimentation this proved about the only method of getting this message displayed in the right place. The point is to give the player some illusion of freedom in conversing with Solomon, but to have the preoccupied king discuss Hiram and the missing bronze bowl no matter what.

Since Solomon introduces these topics into the conversation, it might be as well to add a couple of `AskTopics` in Solomon's `solomonTalking` state to handle them:

```
++ AskTopic, SuggestedAskTopic @bronzeBowl
  "<q>What is this bowl you've lost, exactly?</q> you ask.\b
  <q>It's a vessel for the temple service - a sacred vessel.</q> the
  king tells you, <q>About a cubit in diameter, and decorated with
  rows of pomegranates.</q> "
  name = 'the bronze bowl'
;

++ AskTopic, SuggestedAskTopic @tHiram
  "<q>So why's this Hiram fellow so important?</q> you want to know.\b
  <q>Tyre is a rich and powerful neighbour.</q> Solomon explains, <q>I
  can't afford to offend them. Hiram has been most co-operative in
  supplying material for the building of the Temple, but if he
  thinks I've spurned his gift...</q>"
  name = 'Hiram'
;

tHiram : Topic 'king hiram/tyre';
```

19.9. AgendalItems

19.9.1. AgendalItem

The main purpose of the `AgendalItem` class is to provide a means of giving some goal-seeking behaviour to an NPC by providing NPCs with a list of actions to perform. As the comments in the library code explain it:

Each actor can have its own "agenda," which is a list of these items. Each item represents an action that the actor wants to perform - this is usually a goal the actor wants to achieve, or a conversational topic the actor wants to pursue.

On any given turn, an actor can carry out only one agenda item.

Agenda items are a convenient way of controlling complex behavior. Each agenda item defines its own condition for when the actor can pursue the item, and each item defines what the actor does when pursuing the item.

Note that conversations override agendas. If an actor has an active `ConvNode`, then the actor will not pursue its agenda until it leaves the `ConvNode`. This ensures that the actor's agenda doesn't intrude upon stateful conversations.

A secondary use for `AgendalItems` is simply to provide a way of having an NPC to react to some circumstance without having to override a `takeTurn()` method on its `ActorState` (which may not be particularly convenient if we don't know what `ActorState` the NPC will be in at the time we want the action triggered) or otherwise set up a `Daemon` for testing whether the conditions that would trigger the action have been fulfilled.

For example, if the game reaches a state in which the curator can see the golden banana, this will mean that he has caught the player character in the act of stealing it. We could define an `AgendalItem` (located directly in the curator) to handle this thus:

```
+ stolenBanana : AgendalItem
  initiallyActive = true
  isReady = (getActor.canSee(goldenBanana))
```

TADS 3 Tour Guide

```
invokeItem
{
    "<q>You're trying to steal the Golden Banana of Discord!</q> cries
    {the curator/he}, <q>That's High Treason! Punishable by death!
    Instantly!</q>\b
    So saying he pulls out a revolver and shoots you - twice. But the
    first shot is enough.\b";
    endGame(ftDeath);
}
;
```

Note that this works because this is the *only* Agendaltem we define for the curator. If there were more that might be active at any one time, using an Agendaltem to define the curator's response to seeing the golden banana might not be such a good idea, since we could not be sure that another Agendaltem that happened to be active in the critical turn would not be executed in its stead (although we'll shortly discuss a way round this).

The main properties/methods to note on Agendaltem are:

- **isReady** - this property/method must return true for the Agendaltem to be considered active. By default, this is simply true, but you can override it with a condition that becomes true under certain circumstances, or else start it at nil and then set it to true in some other part of the game code.
- **invokeItem** - the method that is called when this Agendaltem is executed. Override this method to perform whatever actions you want performed when this Agendaltem is invoked.
- **isDone** - when this becomes true the Agendaltem is removed from the actor's list of current Agendaltems. This can be set to a declarative condition (or method) that returns true when certain circumstances obtain, or can be set to true in code (typically by a statement such as `isDone = true` within `invokeItem`).
- **initiallyActive** - Override this to true to make the item initially active; it will then be added to the actor's agenda during the game's initialization.
- **agendaOrder** - The ordering of the item relative to other agenda items. When we choose an agenda item to execute, we always choose the lowest numbered item that's ready to run. You can leave this with the default value (100) if you don't care about the order.
- **getActor()** - returns the actor to which this Agendaltem belongs (effectively returns the location property of the Agendaltem).
- **resetItem()** - resets `isDone` to nil if `isDone` is not a method. This is called whenever an Agendaltem is added to an actor's agendaList, and thus makes it easier to re-use an Agendaltem.

To use an Agendaltem you first need to define it and locate it directly within its actor. To have it taken into account in that actor's turn you then need to add it the actor's agenda, by a call to `actor.addToAgenda(myAgendaltem)`. Each turn when the actor is not otherwise engaged, it will scan its agendaList and call the `invokeItem()` method of the first Agendaltem it finds for which `isReady` is true. It also looks for all the Agendaltems in its agendaList for which `isDone` is true and removes them.

Now suppose we want the curator to react if he sees the player character carrying Solomon's carbuncle (which he wants to acquire for the museum). We could add another Agendaltem like this:

```
+ curatorSurprise : AgendaItem
    initiallyActive = true
    isReady = (getActor.canSee(carbuncle) && carbuncle.isHeldBy(gPlayerChar))
    invokeItem
    {
        "{The curator/he} stares at you in amazement when he sees what
        you're carrying. ";
        isDone = true;
    }
;
```

Note that we only want the curator's reaction to occur once, so we ensure that the `invokeItem` method includes the statement `isDone = true`. The one problem now is that if the player character appears in front of the curator carrying both the Golden Banana of Discord and Solomon's Carbuncle, we can't be sure which of the two Agendaltems will be activated (in fact this could never happen in this game, since the player character cannot gain access to the Golden Banana without first giving the carbuncle to the curator, but the problem could come up in other contexts, so it's worth discussing). The solution to this is to define which Agendaltem we want to take priority, by using the `agendaOrder` property. Since the default `agendaOrder` is 100, we could either lower the order of `stolenBanana` or raise that of `curatorSurprise` (the Agenda item with the lower order takes precedence). We can do the latter, say, by adding `agendaOrder = 150` to the definition of `curatorSurprise`.

19.9.2. ConvAgendaItem

The comments in the library code describe the ConvAgendaItem thus:

A "conversational" agenda item. This type of item is ready to execute only when the actor hasn't engaged in conversation during the same turn. This type of item is ideal for situations where we want the actor to pursue a conversational topic, because we won't initiate the action until we get a turn where the player didn't directly talk to us.

The main thing to note about a ConvAgendaItem is that the only way it differs from a standard AgendaItem is by overriding the isReady property to achieve the desired effect, namely by defining it as:

```
class ConvAgendaItem: AgendaItem
  isReady = (!getActor().conversedThisTurn()
    && getActor().canTalkTo(otherActor)
    && inherited())
;
```

This means that if you want to use your own isReady condition on a ConvAgendaItem you must be careful to include the inherited condition as well:

```
myChatAgenda: ConvAgendaItem
  isReady = inherited && myCustomCondition
  invokeItem()
  {
    /*your code here */
  }
;
```

For our first example of a ConvAgendaItem, we'll simply add the ConvAgendaItem to the actor's agendaList and leave the isReady condition alone. Suppose we want Sarah to react to the news that the main cave exit is blocked; we might define a ConvAgendaItem for her thus:

```
+ sarahWantsOut : ConvAgendaItem
  invokeItem()
  {
    "<q>What I want to know,</q> {the sarah/she} remarks, <q>is how we're
      going to get out of here now that the main exit's blocked.</q> ";
    isDone = true;
  }
;
```

In order for this to be added to her agenda at the right point in the game, we need to make a small change to the TellTopic in which she's told about the rockfall:

```
+ TellTopic, SuggestedTellTopic, StopEventList @tRockfall
[
  {: "<q>We won't get out the way we came it,</q> you warn her, <q>I'm
    afraid there's just been a rockfall, blocking the way back out
    to the outside world!</q> <<sarah.setKnowsAbout(tRockfall)>>\b
    <q>Oh wonderful!</q> {the sarah/she} declares, as if it was your
    fault. <<sarah.addToAgenda(sarahWantsOut)>>" },
  '<q>The way out back through the caves is blocked by a rockfall.</q>
    you say.\b
    <q>Yes, you already told me.</q> she reminds you. '
]
timesToSuggest = 1
name = 'the rockfall'
isActive = entranceTunnel.blocked
;
```

For our second example, we'll turn curatorSurprise into a ConvAgendaItem. While we're at it we'll do the job properly by making it initiate a Conversation Node. Note also how we change the isReady condition from what we had before:

TADS 3 Tour Guide

```
+ curatorSurprise : ConvAgendaItem
  initiallyActive = true
  isReady = (getActor.canSee(carbuncle) && carbuncle.isHeldBy(gPlayerChar) && inherited)
  invokeItem
  {
    getActor.initiateConversation(curatorTalking, 'curator-carbuncle');
    isDone = true;
  }
  agendaOrder = 150
;

+ ConvNode 'curator-carbuncle'
  npcGreetingMsg = "<q>What's that you're carrying?</q> {the curator/he}
    inquires, his eyes lighting up in amazed excitement at the sight of the
    carbuncle, <q>May I take a closer look?</q> ";
;

++ YesTopic
  topicResponse
  {
    "<q>Yes, sure.</q> you agree.\b ";
    nestedAction(GiveTo, carbuncle, getActor());
  }
;

++ NoTopic
  "<q>No - it's not for you.</q> you reply sternly.\b
    <q>Oh please, do show it to me!</q> he begs. "
;
```

One way to try this out if you've included Nikos Chantziaras's ncDebugActions in your project is to SNARF the carbuncle and the cap, PUT the carbuncle in the cap, WEAR the cap, then POW to the large sign. Go WEST from there, engage the curator in conversation, and remove the cap while talking to him.

19.9.3. DelayedAgendaItem

A DelayedAgendaItem, as its name suggests, is an [AgendaItem](#) that becomes active at some point in the future. As with [ConvAgendaItem](#) this relies on an overridden **isReady** method, so that if you want to add your own isReady condition this needs to be combined with the inherited behaviour.

In addition to the properties it inherits from AgendaItem, DelayedAgendaItem has a **readyTime** property which is used to determine when it will become active, and a **setDelay(turns)** method which is used to set readyTime the desired number of turns in the future; the setDelay method returns self (for reasons that will shortly become apparent).

For example, suppose we want a pair of demons to make a threatening advance on the player character (PC) two turns *after* the PC arrives in their vicinity (assuming that the PC remains in their vicinity). The usefulness of the delay in this case is that the demons' sudden outburst is rather less predictable to the player if it doesn't come on the same turn that the player first encounters the demons. First we can define an appropriate DelayedAgendaItem, which should be located in the demons object:

```
+ demonsHissing : DelayedAgendaItem
  isReady = (inherited && getActor.canSee(gPlayerChar))
  invokeItem
  {
    "Two of the demons suddenly spring towards you, hissing wildly and
    pawing at you with their claws. <q>Despair, human, despair!</q> they
    shriek, <q>Your time is running out!</q> ";
    isDone = true;
  }
;
```

Next we want some means of setting this to run two turns after the PC encounters the demons. The best way to do this is probably to add demonsHissing to the demonic agenda in demonsPrancing.afterTravel:

```
++ demonsPrancing : ConversationReadyState
  specialDesc = "A gaggle of demons is prancing around on the rocks near the steps down
    towards the basalt plain, pretending to ignore you, but keeping a careful watch on
```


TADS 3 Tour Guide

```
you just the same. "
stateDesc = "They're prancing around near the steps down to the plain. "
isInitState = true
beforeTravel(traveler, connector)
{
    if(connector == hellPathDown)
    {
        "The demons bunch together and shriek at you, driving you back from the steps. ";
        exit;
    }
    inherited(traveler, connector);
}
afterTravel(traveler, connector)
{
    getActor.addToAgenda(demonsHissing.setDelay(2));
}
;
```

Now we can see the advantage of having `setDelay()` return `self`. This allows us to do in one statement what we should otherwise have to do in two, since

```
getActor.addToAgenda(demonsHissing.setDelay(2));
```

is equivalent to

```
demonsHissing.setDelay(2);
getActor.addToAgenda(demonsHissing);
```

19.9.4. More AgendaItem Examples

Although we've already given several examples of AgendaItems, it may be worth giving a few more to explore what else can be done with this mechanism. All the following examples will apply to Sarah, so should be placed in your code so that they are located directly within the sarah object.

The first problem we'll solve using AgendaItems (though there are doubtless other ways of doing it) concerns the gas masks. So far we have devised a means of giving Sarah a gas mask, but not of making Sarah do anything with it. In order to follow the player character (PC) down the path to the basalt plane in Hell Fire Cavern, Sarah needs to don the gas mask when the PC does. We can use a pair of AgendaItems to make Sarah don her gas mask when the PC is wearing it and remove it again when the PC removes it:

```
+ sarahDonGasMask : AgendaItem
isReady = (gPlayerChar.isMasked)
invokeItem
{
    nestedActorAction(getActor, Wear, getActor.maskObj);
    isDone = true;
    getActor.addToAgenda(sarahDoffGasMask);
}
;

+ sarahDoffGasMask : AgendaItem
isReady = (!gPlayerChar.isMasked)
invokeItem
{
    nestedActorAction(getActor, Remove, getActor.maskObj);
    isDone = true;
    getActor.addToAgenda(sarahDonGasMask);
}
;
```

Note how each of these AgendaItems adds the other to Sarah's agenda when it is invoked. This allows the PC to don and remove his mask as many times as he likes, and Sarah will still follow his lead. At the moment, however, neither AgendaItem starts off in Sarah's agenda, so neither will be executed. Since Sarah starts off without a gas mask, `sarahDonMask` is clearly the AgendaItem that needs to be added first; since Sarah can't wear her gas mask before she's got one, the appropriate place to add this to her agenda is probably when she's first given a gas mask:

TADS 3 Tour Guide

```
+ GiveTopic
  matchTopic(fromActor, obj)
  { return obj.ofKind(GasMask) ? matchScore : nil; }
  handleTopic(fromActor, obj)
  {
    obj.moveTo(getActor);
    "<q>Thanks,</q> {the sarah/she} remarks dubiously as she accepts it from
    you, <q>I'm sure it'll - er - come in very useful.</q> ";
    getActor.addToAgenda(sarahDonGasMask);
  }
;
```

The next example shows how we can combine an AgendaItem with an EventList to provide Sarah with a list of responses to the demons whenever she can see them. It also provides an example of an AgendaItem with a declarative isDone condition.

```
+ sarahHatesDemons : AgendaItem, ShuffledEventList
  isReady = (getActor.canSee(demons))
  initiallyActive = true
  invokeItem
  {
    doScript;
  }
  firstEvents =
  [
    '{The sarah/she} backs away from the demons in disgust.
    <q>Ugh! Get away from me, you horrid creatures!</q>
    she shrieks. '
  ]
  eventList =
  [
    '{The sarah/she} glances in horror at the demons crowding
    around you, raising her arms to ward them off. ',
    'One of the demons clutches at {the sarah\'s/her} arms, and
    she does her best to shake it off. ',
    '{The sarah/she} throws you an anxious glance, <q>Can\'t
    we get out of here?</q> she pleads. ',
    '{The sarah/she} clutches your arm in fright as a pair
    of demons close in on her. '
  ]
  isDone = (demons.isIn(nil)) /* once the demons have been banished they'll never be
                              seen again, so there's no point leaving this in
                              Sarah's agenda list */
;
```

Finally, we can even have Sarah solve one of the puzzles (pulling the burned bush up) if it doesn't occur to the player:

```
+ sarahPullBush : DelayedAgendaItem
  isReady = (getActor.canSee(burnedBush))
  invokeItem
  {
    "{The sarah/she} walks over to the bush and eyes it speculatively. ";
    nestedActorAction(getActor, Pull, burnedBush);
    "<.p>{The sarah/she} peers into the hole. <q>It looks like there's
    some sort of tunnel.</q> she remarks, a little dubiously, <q>I suppose
    you could crawl in there, but I wouldn't fancy it.</q><.p>";
  }
  isDone = (burnedBush.isPulled)
;
```

Again, this needs to be added to Sarah's agenda at some appropriate point:

```
+ bush: CustomImmovable 'dessicated burning bush' 'dessicated bush'
  "The bush is <<daemonID == nil ? 'little more than a collections of
  dried sticks, with only the occasional apology for a leaf doing duty
  for foliage' : 'ablaze'>>. "
  inRoomDesc = "The only sign of life on this barren hillside
  is a dessicated bush. "
  burnDaemon { eventList.doScript; }
```

TADS 3 Tour Guide

```
daemonID = nil
eventList : EventList
{
  [
    'The bush is alight. ',
    'The bush is burning furiously. ',
    'The bush is starting to burn out. ',
    &swap
  ]
  swap()
  {
    "The flames on the bush die out, leaving only charred remains. ";
    lexicalParent.daemonID.removeEvent();
    lexicalParent.daemonID = nil;
    burnedBush.moveTo(lexicalParent.location);
    lexicalParent.moveTo(nil);
    sarah.addToAgenda(sarahPullBush.setDelay(2));
  }
}
cannotTakeMsg = 'Tug as {you/he} will, the bush will not not quite come
  free of the ground. '
;
```

Now, suppose we don't want Sarah to follow the player character into the tunnel - perhaps she doesn't like narrow, dark tunnels running into the side of rumbling volcanos (a pretty understandable attitude when you come to think of it.) The first thing to do is to prevent Sarah following the actor into the hole, which we can do by making the following alteration on the sarahFollowing state:

```
accompanyTravel(leadActor, conn)
{ return leadActor == gPlayerChar && conn != bushHole; }
```

Both to make sure that the player notices that Sarah is no longer following, but waiting on the side of the volcano, and to make Sarah a bit more lifelike, we can then add a couple of AgendaItems to display a "farewell" and "welcome back" message. Note that we need to use callWithSenseContext for the first one so that the text is actually displayed:

```
+ sarahWontFollow : AgendaItem
isReady = (gPlayerChar.isIn(volcanoTunnel) && getActor.curState==sarahFollowing)
invokeItem
{
  callWithSenseContext(nil, nil, {
    "<q>Take care, now!</q> you hear {the sarah/she} call after you
      anxiously.<.p>" });
  isDone = true;
  getActor.addToAgenda(sarahWelcomesBack);
}
;

+ sarahWelcomesBack : AgendaItem
isReady = getActor.canSee(gPlayerChar)
invokeItem
{
  "<q>You're back then!</q> {the sarah/she} declares, with
    evident relief.<.p>";
  isDone = true;
  getActor.addToAgenda(sarahWontFollow);
}
;
```

Finally, we need to get the first of these into Sarah's agenda in the first place. One way we can do this would be to add it, as before, in the swap message of the bush.eventList:

```
+ bush: CustomImmovable 'dessicated burning bush' 'dessicated bush'
...
eventList : EventList
{
  ...
  swap()
  {
    "The flames on the bush die out, leaving only charred remains. ";
    lexicalParent.daemonID.removeEvent();
```

TADS 3 Tour Guide

```
lexicalParent.daemonID = nil;
burnedBush.moveTo(lexicalParent.location);
lexicalParent.moveTo(nil);
sarah.addToAgenda(sarahPullBush.setDelay(2));
sarah.addToAgenda(sarahWontFollow);
}
}
...
;
```

Ideally, one might want to make Sarah's behaviour even more elaborate here, perhaps to have her ask the player character how he fared, depending on whether he carried the golden banana into the hole and whether he emerges with it; but at least this provides a basic framework.

19.10. Commanding NPCs

19.10.1. Overview - Commanding NPCs

From time to time the player may try giving orders to an NPC, in the form SARAH, TAKE THE TORCH or SARAH, CUT THE DISPLAY CASE WITH THE DIAMOND RING. By default the library has NPCs refuse all such commands, but it is possible to override this behaviour.

An NPC's response to a command is, in the first instance, defined in the **obeyCommand(fromActor, action)** method of its current [ActorState](#). If this returns nil, then the actor will refuse the command; if it returns true then the actor will obey the command. For example, if you wanted an actor in a particular ActorState always to obey the commands TAKE, ATTACK and JUMP (but no other commands), you could override obeyCommand on the appropriate ActorState thus:

```
obeyCommand(fromActor, action)
{
    if(action.baseActionClass is in (TakeAction, AttackAction, JumpAction))
        return true;
    else
        return inherited(fromActor, action);
}
```

With a bit more effort, you could also use this method to have the NPC obey the commands, TAKE CARBUNCLE and TAKE BAARAS ROOT but no other command. The trick is knowing how to get at the objects involved in the command. At this point the direct object of the command hasn't been fully allocated to the command, but a list of direct objects is available in the property:

```
action.dobjList_[n].obj_
```

Where *n* is the position in the list (alternatively, you can use `action.getResolvedDobjList[n]`). For a command with a single direct object (e.g. SOLOMON, TAKE THE CARBUNCLE), *n* will be 1. So, if we can assume that the player may type SOLOMON, TAKE THE CARBUNCLE or SOLOMON, TAKE THE BANANA but never SOLOMON, TAKE THE CARBUNCLE AND THE BANANA, we could make Solomon accept the commands, and only the commands, TAKE THE BANANA and TAKE THE CARBUNCLE only by modifying his InConversationState thus:

```
+ solomonTalking : InConversationState
specialDesc = "{The solomon/he} is looking up at you with a thoughtful expression
on his face. "
stateDesc = "He's looking up at you. "
attentionSpan = 20
obeyCommand(fromActor, action)
{
    if(action.ofKind(TakeAction)
        && action.dobjList_[1].obj_ is in (baarasRoot, carbuncle))
        return true;
    else
        return inherited(fromActor, action);
}
;
```

TADS 3 Tour Guide

Actually, if you want to try this (though we don't want it as a permanent part of the game), there's one more change you need to make, otherwise Solomon will stop himself from taking these objects. You'll need to change the start of `solomon.beforeAction` thus:

```
beforeAction()
{
    inherited;
    if(gActionIs(Take) || gActionIs(TakeFrom))
    {
        if(gDobj is in (baarasRoot, carbuncle) && !gDobj.gifted && gActor != self)
        ...
    }
}
```

There are, however, a number of problems with this approach. The first is that for anything much more complicated, the `obeyCommand` method will soon start to resemble tangled spaghetti. The second is that there is no very good way to handle the case where the player issues a command like SOLOMON, TAKE THE CARBUNCLE AND THE BANANA, which will result in both objects being taken. The best we could do is something like this:

```
obeyCommand(fromActor, action)
{
    if(action.ofKind(TakeAction))
    {
        foreach(local cur in action.dobjList_)
        {
            if (cur.obj_ not in (baarasRoot, carbuncle))
                return inherited(fromActor, action);
        }
        return true;
    }
    else
        return inherited(fromActor, action);
}
```

But it's far from ideal.

By now you're maybe wondering if there's anything like the `TopicEntry` mechanism that could come to the rescue here. Well, luckily, there is, for the library defines a [CommandTopic](#) and a [DefaultCommandTopic](#), which, with a bit of judicious tweaking, can both make it easier to program NPCs' responses to commands, and provide a far more powerful mechanism for handling them. But first, we'll start with fairly simple uses of these classes.

19.10.2. CommandTopic

A `CommandTopic` is a kind of `TopicEntry` that can be used to provide an actor's response to a command directed to that actor. At its simplest it merely allows us to customize the message shown when an NPC declines to handle a command. For example, we could decide that Sarah, being a basically non-violent person, will refuse all commands to attack anything or anyone. We can therefore define the following `CommandTopic` for her (which we'll locate in the `sarah` object itself, since Sarah is never violent whatever state she's in):

```
+ CommandTopic, ShuffledEventList @AttackAction
[
    '<q>I\'m really not a violent person.</q> she protests.<.p>'
]
[
    '<q>No, I never resort to violence.</q> she refuses.<.p>',
    '<q>I told you, I don\'t like violence!</q> she reminds you.<.p>',
    '<q>I couldn\'t, I just couldn\'t. </q> she shakes her head.<.p>',
    '<q>No way!</q> she replies. '
]
;
```

With a bit of effort, we can extend this so that a `CommandTopic` matches a specific command directed to a specific object. For example, suppose we want Sarah to make a particular response to SARAH, FOLLOW ME while she's in the `sarahTalking` state. We can make a `CommandTopic` match this by overriding its **matchTopic** method:

TADS 3 Tour Guide

```
++ CommandTopic @FollowAction
  "<q>I'm not going anywhere till I've found my ring.</q> she
  tells you. "
  matchTopic(fromActor, action)
  {
    return action.ofKind(matchObj) && action.dobjList_[1].obj_== fromActor
      ? matchScore : nil;
  }
;
```

We can be quite sure that this will work, since FOLLOW can only take a single direct object; if the player tries typing the command SARAH, FOLLOW THE SHIP AND ME it will simply be rejected.

19.10.3. DefaultCommandTopic

By default, if a player issues a command which the author has not catered for, such as CURATOR, KISS SARAH, the game responds with a message like "The curator refuses your request." The DefaultCommandTopic provides a convenient means of customizing this response, either on each individual actor state, or globally for the actor.

Here's how we might define a global DefaultCommandTopic (located in the sarah object) for Sarah:

```
+ DefaultCommandTopic, ShuffledEventList
[
  '<q>No, I\'d really rather not!</q> she tells you.<.p>',
  '<q>I don\'t think so.</q> she declines.<.p>',
  '<q>Since when did you have the right to boss me around?</q>
  she demands.<.p>',
  '{The sarah/she} merely shakes her head.<.p>',
  '<q>Do it yourself!</q> she tells you.<.p>'
]
;
```

This is fine, except that if a command is directed at Sarah before you've struck up a conversation with her, you get something like this:

>woman, examine the ship

"Hello there," you say.

"Hello." she smiles at you, slightly quizzically.

"Do it yourself!" she tells you.

The player will probably see what it means, but it does read a little strangely. It would be nicer if we could get something like this:

>woman, x ship

"Hello there," you say.

"Hello." she smiles at you, slightly quizzically.

"Young woman, please examine the ship," you request.

"Do it yourself!" she tells you.

This can be achieved, but it will take a bit of work; fortunately this work will prove generally beneficial in extending the power of the CommandTopic system. We'll take it a step at a time.

19.10.4. Overriding obeyCommand

The library version of ActorState.obeyCommand is defined as follows:

```

obeyCommand(issuingActor, action)
{
    /*
     * By default, we ignore all orders. We do need to generate a
     * response, though, so for this purpose, treat the order as a
     * conversational action, with the 'action' object as the topic.
     */

    handleConversation(issuingActor, action, commandConvType);

    /* indicate that the order is refused */
    return nil;
}

```

It is the call to handleConversation that will invoke any CommandTopics (or DefaultCommandTopics). Unfortunately, at this point, the action is not fully resolved. As we have seen, it contains a list of the direct objects (if any) that are the target of the command, but as yet it has not selected any of them (even if there is only one) as the direct object. As it stands, when a CommandTopic is triggered and wants to handle the command on some direct objects but not others, it will have to examine the complete list of direct objects, decide whether it wants to match any of them, and then decide what it wants to do both with the objects that it does match and with those that it doesn't. This makes considerable demands on the author and makes for potentially messy code.

It would be much easier for authors if instead of calling handleConversation once for the complete list of direct objects (where there are direct objects), obeyCommand() called handleConversation once for each of the direct objects in turn. Then all CommandTopics could be written knowing that there will be at most one direct object to any command it handles. If different direct objects of a command are to be handled in different ways, then each object (or set of objects) that is to be handled differently can have a CommandTopic of its own, which is much neater and makes each individual CommandTopic easier to write. In addition, it would be even better if handleConversation set the current direct object for every command passed to handleConversation, so that CommandTopics can use the full range of action methods that assume this is defined.

The code for such an obeyCommand routine is as follows:

```

modify ActorState
    obeyCommand(issuingActor, action)
    {
        /*
         * By default, we ignore all orders. We do need to generate a
         * response, though, so for this purpose, treat the order as a
         * conversational action, with the 'action' object as the topic.
         */

        /*
         * If the action takes a direct object (and so could have a list
         * of direct objects), split it into a series of actions
         * with a single direct object
         */
        local dobjLst = action.getResolvedDobjList;

        if(dobjLst != nil)
        {
            local singleAction = action;

            foreach(local obj in dobjLst)
            {
                singleAction.dobjCur_ = obj;
                handleConversation(issuingActor, singleAction, commandConvType);
            }
        }
        /* Otherwise, just treat it as a single command */

        else
            handleConversation(issuingActor, action, commandConvType);
    }

```

TADS 3 Tour Guide

```
        /* indicate that the order is refused */
        return nil;
    }
;

```

There's really also no need to try to cater for commands with multiple indirect objects, since the standard library defines none, and they are perhaps seldom likely occur; there's very little reason to cater for commands like PUT THE ONION IN THE PAN, THE OVEN AND THE JAR or UNLOCK THE DOOR WITH THE BRASS KEY, THE SILVER KEY AND THE IRON KEY.

19.10.5. TCommandTopic

Having overridden [ActorState.obeyCommand](#) to make it easier for CommandTopics to handle commands with a direct object, the next step is to define a CommandTopic subclass that takes advantage of these changes. We'll call it TCommandTopic, for a topic that handles a TAction.

What we mainly want TCommandTopic to do is to match a direct object (or a list of direct objects) as well as a specific action (or list of direct actions). While we're at it, we'll also make it store a phrase that can be used building a conversational exchange about the command between the actor issuing the command (normally the player character) and the NPC being commanded. Here's what a TCommandTopic might look like:

```
TCommandTopic : CommandTopic
/*
 * The direct object, or a list of direct objects, that will be matched
 * by this topic.
 */
matchDobj = nil

/*
 * The first direct object of the command that this CommandTopic matches.
 * We cache it here so that it can easily be picked up in topicResponse.
 */
currentDobj = nil

/*
 * Cache the action that has been matched so that it is readily accessible
 * from topicResponse
 */
currentAction = nil

matchTopic(fromActor, action)
{
    /* First check whether we match the action of the command */
    if(!inherited(fromActor, action))
        return nil;

    /* Then cache the first direct object of the command */

    currentDobj = action.getDobj();

    /* If matchDobj is a list, check if the current direct object is in the list */
    if(matchDobj.ofKind(Collection))
    {
        if(matchDobj.indexWhich({x: currentDobj.ofKind(x)}) != nil)
            return matchScore;
    }
    else
    {
        /* See if the direct object matches that specified in matchDobj
         * if it's a single object.
         */
        if(currentDobj.ofKind(matchDobj))
            return matchScore;
    }

    /* We can't match the direct object at all, so return nil */
    return nil;
}

```


TADS 3 Tour Guide

```
handleTopic(fromActor, action)
{
    actionPhrase = action.getInfPhrase;
    currentAction = action;

    /*
     * if the player types a command like X ME, getInfPhrase will
     * return 'examine you'. In such a case we want to replace 'you'
     * with 'me'.
     */
    actionPhrase = actionPhrase.findReplace(' you ', ' me ', ReplaceAll);
    if(actionPhrase.endsWith(' you'))
        actionPhrase = actionPhrase.findReplace(' you', ' me', ReplaceOnce,
            actionPhrase.length-5);
    inherited(fromActor, action);
}

/* The action phrase of the command currently directed to this actor;
 * for example, if the player types 'X ME' the actionPhrase will be
 * 'examine me'. This can be used in topicResponse to construct the
 * command given by the player character, e.g.
 *
 *      "Sarah, please examine me," you ask.
 */
actionPhrase = nil
;
```

One thing to note here is that we test for `currentDobj.ofKind(matchDobj)` instead of `matchDobj == currentDobj`. This allows us to specify `matchDobj` as a *class*, or a list of classes, as well as an individual object or list of objects, so we could, for example, easily specify a `TCommandTopic` that matched when the action was a `TakeAction` and the direct object was of the `Treasure` class.

To show a `TCommandTopic` in action, we can now provide Sarah with a suitably coy set of responses to the command, `SARAH, KISS ME`, which we'll attach directly to the actor:

```
+ TCommandTopic, ShuffledEventList @KissAction
[
    '<q>Goodness, no!</q> she declares, <q>What
    <i>do</i> you take me for?</q>'
]
[
    '<q>You are joking, of course!</q> she laughs. ',
    '<q>Ever the optimist, aren\'t we!</q> she chides you. ',
    '<q>Not bloody likely!</q> she retorts. ',
    '<q>Now, why on earth do you suppose I\'d want to do
    that?</q> she cries. ',
    '<q>Don\'t be silly.</q> she admonishes you. '
]
matchDobj = gPlayerChar
;
```

This hardly shows the power of what we've created however. To create a more interesting example, recall that when Sarah and the player character finally come across the Golden Banana of Discord in its display case, the player has to `ASK SARAH FOR RING` to prompt her to cut the case open. We can now use a `TCommandTopic` (located in the `sarahFollowingState`) to enable Sarah also to respond appropriately to the command `SARAH, CUT CASE`:

```
++ TCommandTopic @CutWithAction
matchDobj = bananaCase
topicResponse
{
    "<q>\^<<getActor.name>>, would you <<actionPhrase>>, please?</q>
    you ask her.\b ";
    if(currentAction.iobjList_[1].obj_ != diamondRing)
        "<q>I don't think that\'ll work.</q> she says dubiously. ";
    else
        "<q>Do you really think I should?</q> she asks. <.convnode banana-case>";
}
;
```

With the aid of which we can, for example, generate the following transcript:

TADS 3 Tour Guide

>sarah, cut case with banana

"Sarah, would you cut the display case with the Golden Banana of Discord, please?" you ask her.

"I don't think that'll work." she says dubiously.

>sarah, cut case

"Sarah, would you cut the display case with the diamond ring, please?" you ask her.

"Do you really think I should?" she asks.

(You could say yes or no, or ask what she thinks.)

>no

"No, on second thoughts I think we'd better leave it for now." you reply.

"Very well." she sighs.

Although this exchange could lead to Sarah cutting open the case, we've yet to see an example of a TCommandTopic that leads directly to the NPC performing an action. But it is, in fact, perfectly easy to make any action occur in response to a command using a TCommandTopic. For example, we can easily provide a TCommandTopic that translates SARAH, TAKE THE RING into GIVE RING TO SARAH, which might be relevant while Sarah's still in the sarahTalkingState (i.e., before she starts following the player character around):

```
++ TCommandTopic @TakeAction
  matchDobj = diamondRing
  topicResponse
  {
    "<q>Here, take this,</q> you say, offering her the ring.<.p>";
    nestedAction(GiveTo, diamondRing, sarah);
  }
;
```

We can also, of course, simply have the actor perform the precise command ordered; but note a possible trap here. We might be tempted to define something like this:

```
++ TCommandTopic @TakeAction
  matchDobj = [goldenBanana, hexCrystal]
  topicResponse
  {
    nestedActorAction(sarah, Take, currentDobj);
    "The sarah/she} takes <<currentDobj.theName>> and turns it over in
    her hands. <q>That's interesting!</q> she says.<.p>";
  }
;
```

This may appear to work - unless the nestedActorAction fails for one of a number of reasons (such as either the player character or Sarah holding the golden banana when the command is issued). Then we could end up with a transcript like this:

>sarah, take banana

You won't let Sarah have that. Sarah takes the Golden Banana of Discord and turns it over in her hands. "That's interesting!" she says.

>i

You are carrying the Golden Banana of Discord.

>drop it

Dropped.

>woman, take banana

Sarah takes the Golden Banana of Discord and turns it over in her hands. "That's interesting!" she says.

>woman, take banana

Sarah is already carrying the Golden Banana of Discord. Sarah takes the Golden Banana of Discord and turns it over in her hands. "That's interesting!" she says.

TADS 3 Tour Guide

There are basically two ways to get round this. One is to have the special message display in the `actionDobjTake` handler of the objects concerned, but that could quickly get tedious to code if there were several of them. The other, and, probably easier, is to have the `TCommandTopic` check that the nested `ActorAction` has succeeded before displaying a message that assumes that it has. The trick is to work out what condition best achieves that. In this case, it's probably most effective to test that the object has changed locations as a result of the nested `TakeAction`; if it has, then the nested action must have worked. So we should instead code our `TCommandTopic` thus:

```
++ TCommandTopic @TakeAction
matchDobj = [goldenBanana, hexCrystal]
topicResponse
{
    local oldLoc = currentDobj.location;
    nestedActorAction(sarah, Take, currentDobj);
    /* test that the object has actually moved before reporting that it has */
    if(currentDobj.location != oldLoc)
        "{The sarah/she} takes <<currentDobj.theName>> and turns it over in
        her hands. <q>That's interesting!</q> she says.<.p>";
}
;
```

This will then work correctly. But once you've verified that it does you should comment it out or remove it altogether, since we don't actually want it in the game.

You could go on to define an analogous `TCommandTopic` to cope with commands with two objects, a direct and an indirect, but it is probably not worth the bother. To deal with such a command it is probably easier simply to match the action and the direct object, and then let `topicResponse` handle the matching of the indirect object. The reason for this is that if you're going to have an NPC accept a command involving a particular action and direct object, but only a certain range of indirect objects, it's probably more helpful to the player to have the NPC make some kind of response directly related to the combination of action, direct object and indirect object specified, rather than a generic refusal. For example, if the player types:

>sarah, put the torch in the volcano

It's probably more helpful to have a response like:

"I don't think I should put it there," she replies dubiously.

than a more generic one like:

Sarah refuses your request.

The former response clues the player that Sarah might be prepared to put the torch somewhere, but just not there.

Note that if you think something like `TCommandTopic` might be useful in your own game, you can download a (rather more complete) `TCommand` library extension from the IF-Archive.

19.10.6. A Modified `DefaultCommandTopic`

Some way back, we promised to give a modified form of the `DefaultCommandTopic` that would generate a more satisfactory transcript when a command is issued to Sarah before she's in her `InConversationState` (so that the greeting protocols intervene between the issuing of the command and Sarah's response to it). In the light of the changes made up to this point, we should now modify `DefaultCommandTopic` thus:

```
modify DefaultCommandTopic
handleTopic(fromActor, action)
{
    actionPhrase = action.getInfPhrase;

    /*
    * if the player types a command like X ME, getInfPhrase will
    * return 'examine you'. In such a case we want to replace 'you'
    * with 'me'.
    */
```

TADS 3 Tour Guide

```
    actionPhrase = actionPhrase.findReplace(' you ', ' me ', ReplaceAll);
    if(actionPhrase.endsWith(' you'))
        actionPhrase = actionPhrase.findReplace(' you', ' me', ReplaceOnce,
            actionPhrase.length-5);
    currentAction = action;
    inherited(fromActor, action);
}
actionPhrase = nil
currentAction = nil
;
```

We can then redefine Sarah's DefaultCommandTopic thus:

```
+ DefaultCommandTopic
topicResponse()
{
    "<q>\^<<getActor.name>>, please
    <<actionPhrase>>, </q> you request.<.p>";
    responseList.doScript();
}
responseList : ShuffledEventList
{
    [
        '<q>No, I\'d really rather not!</q> she tells you.<.p>',
        '<q>I don\'t think so.</q> she declines.<.p>',
        '<q>Since when did you have the right to boss me around?</q>
        she demands.<.p>',
        '{The sarah/she} merely shakes her head.<.p>',
        '<q>Do it yourself!</q> she tells you.<.p>'
    ]
}
;
```

And we now can indeed get transcripts like the following:

>woman, x ship

"Hello there," you say.

"Hello." she smiles at you, slightly quizzically.

"Young woman, please examine the ship," you request.

"Do it yourself!" she tells you.

20. Consultables

20.1. Consultable

A Consultable is something you can look things up in, in response to commands like LOOK BANANA UP IN BOOK or CONSULT BOOK ABOUT MEANING OF LIFE. It follows that a Consultable will normally be something like a book, or maybe a computer. The advantage of a Consultable is that it can contain [ConsultTopics](#), which work in much the way as other [TopicEntries](#) work for an actor.

We'll define just one Consultable in this game: a book, which we'll also make a Readable:

```
historyBook : Readable, Consultable 'dark blue book' 'dark blue book' @glassShelf
    "The gold-lettered title on the dark blue cover is <i>The Great History</i>."
    readDesc = "It's too long to read from cover to cover right now, but
        you could consult it on specific topics. "
    dobjFor(Read)
    {
        preCond = inherited + objHeld;
    }
    dobjFor(Consult)
    {
        preCond = inherited + objHeld;
    }
;
```

Just to make things a little more difficult, we've put the history book inside a glass-fronted shelf, so that it can only be read by first opening the front of the shelf. To enforce this we add objHeld to the list of preconditions for both reading and consulting the book. The shelf front is unlocked by pushing a button, but this will only work if the shelf is plugged in to the electrical supply. To do this requires the power cable that's also used for the vending machine. We'll start this cable off in a wall-mounted cabinet in a storeroom next to the library, but to reach the cabinet you need to stand on something, so we'll put a convenient stool in the library for the purpose. Since the doors to the library and the storeroom are next to each other in the south wall, we'll need to use an AskConnector.

None of this is new, but it may refresh your memory of things we've used before:

```
stoneLanding : Room 'Landing' 'the landing'
    "A pair of doors lead south from this narrow landing, from which
        a narrow flight of stone steps lead down to the north. "
    down = slStairsDown
    north asExit(down)
    south : AskConnector
    {
        promptMessage = "There are two doors to the south. "
        travelAction = GoThroughAction
        travelObjs = [leftDoor, rightDoor]
        travelObjsPhrase = 'of them'
    }
;

+ slStairsDown : StairwayDown ->eastShoreUp 'flight stone down stairs/steps'
    'stone steps down'
    isPlural = true
;

+ leftDoor : Door 'left hand door*doors' 'left hand door'
;

+ rightDoor : Door 'right hand door*doors' 'right hand door'
;

library : Room 'Library' 'the library'
    The library is a long rectangular room, with bookshelves all around.
    It looks, however, as if the shelves have been raided, for they are
    all bare, apart from a glass fronted-shelf at the southern end of
    the room. Not far from this shelf is an electrical socket in the wall. "
    north = libraryDoor
```

TADS 3 Tour Guide

```
out asExit(north)

;

+ libraryDoor : Door -> rightDoor 'door' 'door'

;

+ Decoration '(book) shelves/bookshelves' 'bookshelves'
"The bookshelves are all bare, apart from the glass-fronted shelf
at the end. "

;

+ glassShelf : IndirectLockable, OpenableContainer, Fixture
'glass fronted glass-fronted (book) shelf/front' 'glass-fronted shelf'
"The glass-fronted shelf is effectively a container for one or two books.
A small brown button is mounted on one side, next to an electrical inlet. "
material = glass
powerOn = (powerCable.isAttachedTo(glassSocket) && powerCable.isAttachedTo(librarySocket))
iobjFor(PutOn) asIobjFor(PutIn)
afterAction()
{
  if(gActionIs(AttachTo) && powerOn && !isLit)
  {
    "A light comes on inside the glass-fronted shelf. ";
    isLit = true;
  }
  if(gActionIs(DetachFrom) && !powerOn && isLit)
  {
    "The light in the glass-fronted shelf goes out. ";
    isLit = nil;
  }
}
isLit = nil
brightness = (isLit ? 3 : 0)

;

+ NameAsOther, SecretFixture
targetObj = glassShelf

;

++ Button, Component 'small brown button' 'small brown button'
dobjFor(Push)
{
  action()
  {
    if(glassShelf.powerOn)
    {
      "A small click comes from the glass-fronted shelf. ";
      glassShelf.makeLocked(!glassShelf.isLocked);
    }
    else
      "Nothing happens. ";
  }
}

;

++ glassSocket : PlugAttachable, Attachable, Component 'electrical inlet'
'electrical inlet'
"The electrical inlet is mounted next to the brown button on the
side of the glass-fronted shelf. "

;

+ librarySocket : PlugAttachable, Attachable, Fixture 'electrical (wall) socket'
'wall socket'
"The electrical socket is mounted on the wall, a couple of inches
up from the floor. It's just a standard socket. "

;

+ stool : Chair 'reading stool' 'reading stool'
"It's a plain wooden stool, without a back. "
initSpecialDesc = "A plain wooden stool stands in the middle of the room. "

;
```

TADS 3 Tour Guide

```
storeRoom : DarkRoom 'Store Room' 'the store room'
    "The storeroom is completely bare apart
    from a cabinet mounted on one wall. "

    north = storeRoomDoor
    out asExit(north)
;

+ storeRoomDoor : Door ->leftDoor 'door' 'door'
;

+ storeRoomCabinet : OutOfReach, OpenableContainer, Fixture
    'cabinet' 'cabinet'
    "The cabinet is mounted high up on the west wall. "
    canObjReachContents(obj)
    { return obj.location.ofKind(Chair); }

;

++ powerCable : PlugAttachable, Attachable, Thing
    'thick black power cable/cord/lead/plugs' 'black cable'
    "It's a thick black power cable, about four feet long, with
    plugs both ends. "
    bulk = 2
    canAttachTo(obj)
    {return obj is in (socket, vendingMachine, librarySocket, glassSocket); }
    travelWhileAttached (movedObj, traveler, connector)
    {
        if(movedObj==self)
        {
            foreach(local cur in attachedObjects)
                tryImplicitAction(DetachFrom, self, cur);
        }
    }
;


```

Note that this is the same power cable we defined before: we've simply moved it to a new initial location and added a couple of items to the list of things it can be attached to.

20.2. ConsultTopic

A ConsultTopic is the class of [TopicEntry](#) that is used with a [Consultable](#). Just as an [AskTopic](#), for example, provides a response to commands such as ASK CURATOR ABOUT MUSEUM, so a ConsultTopic would provide a response to CONSULT HISTORY BOOK ABOUT MUSEUM or LOOK UP MUSUEM IN BOOK.

Although ConsultTopic inherits all the methods and properties of TopicEntry, there is usually less reason to override any of them. A typical Consultable, such as a book, is a fairly static object that will always provide the same information on any given topic, regardless of the state of the game. You could, perhaps, combine a TopicEntry with an [EventList](#) class to provide a series of responses, perhaps representing the player character reading further on the same topic, particularly if you want your Consultable to provide a large amount of information on the topic and it is better broken up into smaller chunks; but then you'd probably want to make it abundantly clear that there *was* more to read on a given topic rather than leaving the player to guess. Again, you could have a more complex Consultable, such a computer, where the information displayed might change with circumstances, in which case you might want to use some of the techniques we have already seen in connection with TopicEntries on actors (such as [AltTopics](#) and the use of isActive).

In the present game, however, we'll keep things at their simplest and just add a number of basic ConsultTopics to our history book:

```
+ ConsultTopic [ghost, statue, tBenedict]
    "According to the book, Benedict the Banana-Bearer lived 537 years ago
    and was responsible for bearing the Golden Banana of Discord from
    Hell Fire Cavern. Although many at the time (and since) reckoned
    that the ordeal drove him mad, this was no obstacle to his becoming
```

TADS 3 Tour Guide

```
king, for many reckoned his deed to be a feat of great daring. A golden
statue commemorating the glorious memory of King Benedict can still be
found on the western side of the lake.
<<gSetKnown(goldenBanana)>>"
;

+ ConsultTopic @goldenBanana
"The Golden Banana of Discord is a talisman of great power. Snatched
from the demons of hell fire cavern some 537 years ago it enabled
mad Benedict to become king. No one since King Benedict has been
quite sure what powers the Golden Banana wields, but its value and
significance remain unquestioned. For obvious reasons great care
has always been taken to ensure that this artifact does not fall into
the wrong hands. "
;

+ ConsultTopic @tHellFireCavern
"Deep in the bowels of the earth, Hell Fire Cavern is a name that has
long struck fear and dread into the souls of mortals. Inhabited by vile
demons, and poluted by the putrid flames of Mount Gloom, the cavern is
as near to being an authentic underworld as one could hope - or fear -
to encounter. Only one man - Benedict the Banana-Bearer - has been known
to return from the Hell Fire Cavern alive. "
;

+ ConsultTopic @tMuseum
"According to the book, the Museum of Curious Antiquities was set up from
a benefaction of King Benedict the Banana-Bearer himself. Although the
museum has always had a certain historical interest, its exhibits tend
towards the eccentric and the bizarre. Should you find yourself on the
southern side of the lake, however, the museum is well worth a visit
if only for its curiosity value. "
;

tBenedict : Topic 'benedict';
tHellFireCavern: Topic 'hell fire cavern';
```

Note the use of the list of matchable topics in the first ConsultTopic, since the player might become aware of Benedict through either the statue or the ghost. Again, since the statue does not match the word 'benedict' and the ghost may not either, we also supply a tBenedict topic to ensure that the word 'benedict' will always match this ConsultTopic (along with such other words as 'statue' and 'ghost' that may also do so).

Note also that there's no such thing as a SuggestedConsultTopic. With a suitable amount of ingenuity it might be possible to devise a SuggestedConsultTopic class that worked, but this would be non-trivial, since the library currently assumes that SuggestedTopics work only with actors. If you want to suggest to the player what topics might be available in the book, it might be simpler to list them in the readDesc (e.g. ""It's too long to read from cover to cover right now, but you could consult it on specific topics, such as Benedict, the Golden Banana of Discord, the museum and Hell Fire Cavern"), or else perhaps in the [DefaultConsultTopic](#), (e.g. "The book doesn't seem to have anything useful to say in that, but a quick check in the index suggests you could look up..."). We'll return to the second possibility shortly.

20.3. DefaultConsultTopic

If the player tries to consult our History Book about something for which we've defined no response, we need an appropriate message to be displayed. We use a DefaultConsultTopic for this purpose (just as we use other [DefaultTopic](#) types in conversation).

An appropriate DefaultConsultTopic is not hard to define. Since a book is an inanimate object, we don't need to vary its behaviour, and something simple like the following should suffice:

```
+ DefaultConsultTopic
"The book doesn't seem to have anything useful to say on that subject. "
;
```

If you wanted to make this a bit more sophisticated by suggesting to the player the topics that are available to be listed in the book, you could add a name property to each ConsultTopic you want suggested (e.g. name = 'Benedict' or name='the museum') and then expand the DefaultConsultTopic thus:

TADS 3 Tour Guide

```
+ DefaultConsultTopic
"The book doesn't seem to have anything useful to say on that subject,
but a quick look in the index suggests that you could consult it
about <<suggestionList>>. "
suggestionList()
{
    local lst = [];
    foreach(local cur in location.consultTopics)
        if(cur.name != nil)
            lst += cur.name;
    stringLister.showList(lst); // n.b., not in the library but a custom object defined above
}
;
```

There are both advantages and disadvantages to this approach. On the one hand the player is not left to guess which topics the book implements; on the other, the message is a bit directive and may make consulting the book feel like working through the list of topics suggested.

A less directive approach, which may provide the best compromise, could be to provide the book with an index topic and drop a broad hint in the DefaultConsultTopic that there is an index to be consulted:

```
+ ConsultTopic @tIndex
"A quick look in the index suggests that you could consult it
about <<suggestionList>>. "
suggestionList()
{
    local lst = [];
    foreach(local cur in location.consultTopics)
        if(cur.name != nil)
            lst += cur.name;
    stringLister.showList(lst);
}
;
```

```
+ DefaultConsultTopic
"So far as you can tell from the index, the book doesn't seem to
have anything useful to say on that subject. "
;
```

```
tIndex : Topic 'index/contents';
```

If you had several books in your game that you wanted to provide with an index you could avoid repetitive coding by defining an IndexTopic class:

```
class IndexTopic : ConsultTopic
    suggestionList()
    {
        local lst = [];
        foreach(local cur in location.consultTopics)
            if(cur.name != nil)
                lst += cur.name;
        stringLister.showList(lst);
    }
;
```

21. Scoring

21.1. Scoring - Overview

In some, though not all, games you may want to keep the score, awarding points for various achievements, and notifying players how they're getting on by showing their current score in relation to the maximum score available. In addition you may want to assign a descriptive rank (e.g. from 'novice' to 'hero') to the player depending on the score currently attained. In addition, you may want players to be able to see how their score is made up, with a list containing the scores for individual achievements, and aggregate scores for related achievements (such as finding the different members of a set of related items).

The TADS 3 library allows you to do all this fairly simply. You can change the player's score with a simple [addToScore](#) statement at any point, or use [Achievement](#) objects to help keep track of the point they've earned. For aggregating scores for similar feats under a common description, the [SimpleAchievement](#) class can be useful. Finally, you can set up a [scoreRankTable](#) to translate scores into descriptive ranks (e.g. "This makes you a mere blundering novice/fantastic superhero") and define the [maximum score](#) attainable.

21.2. addToScore

The simplest (in some respects) way to keep track of the score is to call the **addToScore** function, which is called with two arguments: `addToScore(points, description)`, where *points* is the number of points being awarded and *description* is a single-quoted string describing why the points are being awarded. For example, to award a single point for unlocking the door from anotherCave to lakeRoom (by inserting the brass coin) you can simply call `addToScore` in the `notifyInsert` method of the slot:

```
++ RestrictedContainer, Component 'small vertical slot' 'slot'
    "It's about half an inch long; next to it is some faded writing that
    just about still says\nENTRANCE TO LAKE\nONE GROAT. "
    validContents = [silverCoin, brassCoin]
    notifyInsert(obj, newCont)
    {
        if(obj==brassCoin)
        {
            "As the brass coin disappears into the slot you hear a click from the door. ";
            obj.moveInto(nil);
            lakeDoor.makeLocked(nil);
            addToScore(1, 'unlocking the door to the lakeside ');
        }
        else
        {
            "Despite initial appearances <<obj.theName>> doesn't seem to be quite
            right for the slot. ";
        }
        exit;
    }
;
```

Note that the score will be incremented every time `addToScore` is called, so if we want to make sure that points are awarded only once for a particular achievement, we must make sure that the corresponding `addToScore` is called only once. In this particular case we are safe: since there's only one brass coin in the game and it disappears forever when it's put in the slot, this `addToScore` can only ever be executed once.

Blowing up the boulder is similar situation: since this can only ever occur once we can quite safely put an `addToScore` in the dynamite object's `sayBurnedOut()` method (insofar as playing with dynamite can ever be said to be safe!). We might want to award more points for this:

```
dynamite : Candle 'stick dynamite/fuse' 'stick of dynamite'
    "It's a white cylinder with a short fuse. <<isLit ?
    'The fuse is lit and burning down fast. ' : nil >>"
    fuelLevel = 3
    brightnessOn = 1
    sayBurnedOut()
```

TADS 3 Tour Guide

```
{
  if(isHeldBy(gPlayerChar))
  {
    "The dynamite explodes with a mighty bang and blows your hand off. But
    since you're killed by the blast you probably won't be needing it
    any more.\b";
    endGame(ftDeath);
  }
  if(canBeTouchedBy(gPlayerChar))
  {
    "The dynamite denonates close by, but you are killed by the blast almost
    before you hear the bang. ";
    endGame(ftDeath);
  }
  if(isIn(boulder))
  {
    boulderFragments.moveInto(boulder.location);
    boulder.moveInto(nil);
    addToScore(2, 'blowing up the boulder');
  }
  callWithSenseContext (nil, nil, {"You hear a muffled explosion nearby. "});
  moveInto(nil);
  fuseID = nil;
  fuelLevel = 3;
}
;
```

Indeed, we can even use this technique for awarding five points for casting the Golden Banana of Discord into Mount Gloom; since this results in moving the Golden Banana into nil and it's never recoverable thereafter, we can put the relevant `addToScore` call in the banana's `moveInto` method:

```
++ goldenBanana : Thing 'golden banana/discord' 'Golden Banana of Discord'
  "It's about the shape and size of an ordinary banana, but seems to be made
  of solid gold. "
  aName = (theName)
  weight = 6
  isListedInContents = (!isIn(bananaCase))
  moveInto(newCont)
  {
    inherited(newCont);
    if(newCont==nil)
      addToScore(5, 'destroying the Golden Banana in Mount Gloom');
  }
;
```

Finally, note that the second parameter in `addToScore` may be an [Achievement](#) object instead of a single-quoted string. We'll explain this in more detail once we've introduced the Achievement class.

21.3. Achievement

Using `addToScore` is fine where we can easily make sure that it will be called only once, but in some situations this becomes harder to guarantee.

For example, the player should probably be rewarded for getting the door into the museum corridor open, since this is a puzzle that takes several steps to solve, but if we just put `addToScore` in, say, `blankSteelDoor.makeOpen()`, then players could keep adding to their score by repeatedly opening the blank steel door once they'd worked out how to [fix the button](#). We *could* prevent this by adding a `hasBeenOpened` property to the `blankSteelDoor`, setting it to true when the door is opened, and only calling `addToScore` if `hasBeenOpened` is nil. But it would be nice if there was an `addToScoreOnce` function that effectively did this for us. Well, there's no such *function*, but there is always the **`addToScoreOnce(points)`** *method* of the Achievement class, and we could always use that.

This does mean, of course, that we need to set up an Achievement object for the purpose, but there's no reason why this shouldn't simply be an anonymous nested object, so that, for example, we could get the result we wanted by modifying `blankSteelDoor` thus:

TADS 3 Tour Guide

```
+ blankSteelDoor : Door 'blank steel door' 'blank steel door'
"The door <<isOpen ? 'has slid open out of sight' : 'is without handle,
keyhole or any other visible mechanism'>>"
openStatus { }
checkDobjOpen() { "There's nothing on the door to get a purchase on. "; exit; }
checkDobjClose() { "The door has slid out of sight. "; exit; }
makeOpen(stat)
{
    inherited(stat);
    achievement.addToScoreOnce(3);
    "The steel door slides <<isOpen ? 'open' : 'shut'>>. ";
}
achievement : Achievement { "opening the blank steel door" }
;
```

This will increase the player's score by three points the first time the player manages to get the blank steel door open, but not thereafter.

At this point it may be helpful to take a closer look at the properties and methods the Achievement class defines:

- **desc** - Describe the achievement - this must display a string explaining the reason the points associated with this achievement were awarded. Note that this description can make use of the scoreCount information to show different descriptions depending on how many times the item has scored. For example, an achievement for finding various treasure items might want to display "finding a treasure" if only one treasure was found and "finding five treasures" if five were found. In some cases, it might be desirable to keep track of additional custom information, and use that information in generating the description. For example, the game might keep a list of treasures found with the achievement, adding to the list each time the achievement is scored, and displaying the contents of the list when the description is shown.
- **scoreCount** - The number of times the achievement has been awarded. Each time the achievement is passed to addToScore(), this is incremented. Note that this is distinct from the number of points.
- **totalPoints** - the number of points awarded for the achievement; if this achievement has been accomplished multiple times, this reflects the aggregate number of points awarded for all of the times it has been accomplished.
- **addToScoreOnce(points)** - Add this achievement to the score one time only. This can be used to score an achievement without separately tracking whether or not the achievement has been accomplished previously. If the achievement has already been scored before, this will do nothing at all; otherwise, it'll score the achievement with the given number of points.
- **points** - the number of points awarded for this Achievement if either its [awardPoints](#) or its [awardPointsOnce](#) method is called.
- **maxPoints** - the maximum number of points that can be awarded for this Achievement if its points can be awarded for it more than once. By default this is simply the same as *points*. This figure is used by the library routine that calculates the [maximum score](#).
- **awardPoints()** - adds this Achievement to the list of Achievements accomplished, and awards the number of points contained in its *points* property (which *must* then be overridden to something other than nil).
- **awardPointsOnce()** - this is the same as awardPoints, except that the points are awarded only the first time this method is called (so that you would typically use this method like addToScoreOnce, which it in fact calls, to ensure that the player is not awarded points multiple times for the same achievement).

You'll see from this that an Achievement object is geared up to the possibility of awarding points for the same achievement more than once. The situation where you'd mostly likely want to do this is in aggregating points for similar feats under a common generic description, such as:

23 points for finding sundry items

Rather than listing the twenty-three sundry items separately at one point each.

In the present game, we might want to define such an aggregating achievement for getting past sundry minor obstacles:

```
obstacleAchievement : Achievement
desc = "getting past <<scoreCount > 1 ? spellInt(scoreCount) +
' sundry obstacles' : 'an obstacle'>>. "
;
```

Then we can employ this obstacleAchievement at each place in the code where we want to reward the player for getting past a minor obstacle, by calling addToScore(*points*, obstacleAchievement); though now we have to make

TADS 3 Tour Guide

sure that we call this only once for each obstacle; for example:

```
secretPassage : Room 'Secret Passage' 'the secret passage'
  "This hitherto secret passage narrows to a long tunnel running north. To the
  south <<rock2.isOpen ? 'an opening leads out into a large, ruddy-hued cave'
  : 'a large rock blocks the way out'>>. "
  south = rock2
  north = tunnel
  brightness = (rock2.isOpen ? 3 : 0)
  enteringRoom(traveler)
  {
    if(!seen) addToScore(1, obstacleAchievement);
  }
;

crewQuarters : DarkCabin 'Crew Quarters' 'the crew quarters'
  ...
  enteringRoom(traveler)
  {
    darkEvents.curScriptState = 1;
    if(!seen) addToScore(1, obstacleAchievement);
  }
;

hellPath : OutdoorRoom 'Path down Hell Fire Cavern' 'the path'
  ...
  enteringRoom(traveler)
  { if(!seen) addToScore(2, obstacleAchievement); }
;

chasmLedge : DarkRoom 'Ledge of Chasm' 'the ledge of the chasm'
  "A deep, wide chasm splits the ground immediately to the north of this
  narrow ledge, while a dark tunnel runs south. Another tunnel can be
  seen leading north from the far side of the chasm. "
  north = deepChasm
  south = tunnelFromChasm
  inRoomName(pov)
  {
    return 'on the ' + (pov.isIn(deepChasm) ? 'south' : 'far') + ' ledge of the chasm';
  }
  enteringRoom(traveler)
  { if(!seen) addToScore(1, obstacleAchievement); }
;

museum : Room 'Museum of Curious Antiquities' 'the main museum'
  ...
  enteringRoom(traveler)
  { if(!seen) addToScore(1, obstacleAchievement); }
;

templeCellar : DarkRoom 'Cellar beneath Temple' 'the cellar beneath the Temple'
  "This long, damp cellar probably hasn't been visited in years. "
  ...
  enteringRoom(traveler)
  { addToScore (1, obstacleAchievement); }
;
```

Of course some of the obstacles are more noteworthy and would definitely deserve their own individual achievements, along the lines we have already seen, such as:

```
+ Lever, Component 'banana-shaped banana shaped projection' 'banana-shaped projection'
  "Protruding from the north side of the altar, the banana-shaped projection is
  its only decorative feature. "
  makePulled(pulled)
  {
    if(stoneAltar.getWeight != 54)
    {
      reportFailure('It won\'t budge. ');
      exit;
    }
  }
```

TADS 3 Tour Guide

```
    else if(pulled)
        "With a loud grating sound, the wall behind the altar grinds open. ";
    else
        "When you push the lever, the wall behind the altar grinds shut. ";
    inherited(pulled);
    templeWestWall.makeOpen(pulled);
    achievement.addToScoreOnce(5);
}
weight = 0
achievement : Achievement { "opening the wall behind the altar " }
;
```

Note that we have used the [Achievement Template](#) in these definitions.

21.4. SimpleAchievement

SimpleAchievement is a subclass of Achievement with a construct method added. This allows you to pass a string in a new SimpleAchievement statement to be used as the description of the achievement for which points are being awarded. This could be useful, for example, if we wanted to award a point for finding each of the tablets. Instead of coding this separately on each tablet object, we can write the code once on the Tablet class. We'll award the points once when each tablet is taken; to do this we override the actionDobjTake method to add a call to addToScoreOnce(). But we want this called on a separate Achievement (or rather, SimpleAchievement) object for each separate tablet. We can't use the anonymous nested object syntax for that; if we wrote:

```
class Tablet : Readable
    desc = "\^<<theName>> is about eight inches square and an inch thick.
    <<readDesc>>"
    readDesc = "On it is inscribed:\b<FONT FACE='TADS-Typewriter'><<inscription>></FONT>\b"
    bulk = 4
    achievement: Achievement { desc = "finding <<theName>>" } // this won't work
;
```

We should end up with only one Achievement object for the whole class, which wouldn't work at all the way we want it to. Instead we need to use the **perInstance** macro to create a new instance of an Achievement object for each instance of a Tablet object. But then we need to find a way to make each Achievement object describe itself appropriately; using the SimpleAchievement class, which allows us to pass a description string in its constructor (i.e. as a parameter in a new SimpleAchievement statement) provides just the solution we need:

```
class Tablet : Readable
    desc = "\^<<theName>> is about eight inches square and an inch thick.
    <<readDesc>>"
    readDesc = "On it is inscribed:\b<FONT FACE='TADS-Typewriter'><<inscription>></FONT>\b"
    bulk = 4
    achievement = perInstance(new SimpleAchievement('finding ' + theName)) // this works just fine
    dobjFor (Take)
    {
        action()
        {
            inherited;
            achievement.addToScoreOnce(1);
        }
    }
;
```

21.5. awardPoints

As an alternative to calling [addToScore](#) or [addToScoreOnce](#), you can call [awardPoints](#) or [awardPointsOnce](#) on an Achievement object. This then automatically awards the number of points associated with the Achievement (i.e. defined in the points property of the Achievement object). For example, to have the player awarded 2 points for disposing of the demons, we might change the appropriate ShowTopic thus:

TADS 3 Tour Guide

```
++ ShowTopic @baarasRoot
topicResponse
{
  "As you produce the baaras root and hold it up before their demonic little
  eyes, it starts to glow an eerie pink colour...
  ...
  ... they
  shimmer and dissolve, turning into plumes of oily black smoke which
  vanishes like a mist. ";
  demons.moveTo(nil);
  achievement.awardPoints();
}
achievement : Achievement { +2 "exorcizing the demons" }
;
```

Note that since the demons can only ever be exorcized once, we don't need any checks to prevent the points being awarded several times over. Otherwise, it would be more convenient to use [awardPointsOnce](#). Note also the use of the [Achievement template](#).

So why do we need awardPoints and awardPointsOnce in addition to addToScore and addToScoreOnce? There are two implications here:

1. With awardPoints and awardPointsOnce the Achievement object determines the number of points awarded. With addToScore and addToScoreOnce the function/method call determines the number of points awarded.
2. Which set of methods you use can have implications for the calculation of the [maximum score](#).

21.6. awardPointsOnce

In cases where you want to prevent points being awarded several times for the same feat, you could use awardPointsOnce in preference to [awardPoints](#) and as an alternative to [addToScoreOnce](#).

```
+++ bronzeBowl : Container 'bronze bowl/vessel' 'bronze bowl'
  "It's of ancient and somewhat curious design, cast with two rows of
  pomegranates all the way round the outside. "
  bulkCapacity = 4
  bulk = 5
  weight = 5
  dobjFor (Take)
  {
    action()
    {
      achievement.awardPointsOnce();
      inherited;
    }
  }
  achievement : Achievement { +3 "recovering the bronze bowl" }
;
```

Once again, note the use of the [Achievement template](#) here.

21.7. scoreRankTable

Many games not only display a score, they translate that score into a rank, along the lines of

"Your score is 0 of a possible 10, in 0 moves. This makes you a tourist. "

To get TADS 3 to do this you need to set up a **scoreRankTable** on the **gameMain** object, for example:

TADS 3 Tour Guide

```
gameMain: GameMainDef
    initialPlayerChar = me
    scoreRankTable =
    [
        [ 0, 'a tourist'],
        [ 5, 'a novice explorer'],
        [ 10, 'a willing amateur'],
        [ 15, 'an apprentice adventurer'],
        [ 25, 'an accomplished adventurer'],
        [ 40, 'a hero']
    ]
;
```

Note how this works: the `scoreRankTable` is a list, each element of which is itself a list of two elements, a number and a single-quoted string. The number is the minimum score needed to reach the rank described in the string. These two-element sublists *must* be arranged in ascending order of score.

21.8. maxScore

If you want to set the maximum score in your game manually, you will need to override `gameMain.maxScore`:

```
gameMain: GameMainDef
    initialPlayerChar = me
    scoreRankTable =
    [
        [ 0, 'a tourist'],
        [ 5, 'a novice explorer'],
        [ 10, 'a willing amateur'],
        [ 15, 'an apprentice adventurer'],
        [ 25, 'an accomplished adventurer'],
        [ 40, 'a hero']
    ]
    maxScore = 43
;
```

However, it is possible to have the game calculate the maximum score automatically, provided you follow certain rules in the way you award points. These rules, as they are described in the library code comments, are as follows:

You can use EXCLUSIVELY Achievement objects to represents scoring items, and give each Achievement object a 'points' property indicating the number of points it's worth. To award a scoring item, you call the method `awardPoints()` on an Achievement object. If you use this style of scoring, the library AUTOMATICALLY computes the `gameMain.maxScore` value, by adding up the 'points' values of all of the Achievement objects in the game. For this to work properly, you have to obey the following rules:

- *use ONLY Achievement objects (never strings) to award points;*
- *set the 'points' property of each Achievement to its score;*
- *define Achievement objects statically only (never use 'new' to create an Achievement dynamically)*
- *if an Achievement can be awarded more than once, you must override its 'maxPoints' property to reflect the total number of points it will be worth when it is awarded the maximum number of times;*
- *always award an Achievement through its `awardPoints()` or `awardPointsOnce()` method;*
- *there exists at least one solution of the game in which every Achievement object is awarded*

We have not followed these rules in this game (otherwise we should not have been able to demonstrate the other ways of awarding points), so this cannot be demonstrated here. Note, however, that there *may* be a way of sorts round the last two restrictions in some cases. In particular, if there are alternative routes through the game so that points can be awarded (say) for doing either A or B but not both, then you could award points for A using `awardPoints()` or `awardPointsOnce()`, and points for B by using `addToScore` or `addToScoreOnce`, and provided the same number of points are awarded for A or B, then the automatic maximum score calculation should still work.

You also need to bear in mind the following rules about setting `maxScore`:

TADS 3 Tour Guide

- If you do not explicitly override `gameMain.maxScore` at all in your game code, then the game will automatically calculate the maximum score on the basis of the rules given above.
- If you explicitly set `gameMain.maxScore` to `nil`, then the game will assume there is no defined maximum score, and the maximum score will not be mentioned at all in response to the `SCORE` and `FULL SCORE` commands.
- If you explicitly set `gameMain.maxScore` to a number, that number will be used as the maximum score.

22. Hints

22.1. Hints - Overview

If you want to add a context-sensitive hints system to your game, it's probably a lot of work, but at least the TADS 3 library does what it can to help by providing a set of classes to ease the coding of a menu-based hint system. Basically, all you need to do is to set up a menu tree that, in outline, might look something like this:

```

TopHintMenu 'Hints';

+ HintMenu 'Cave Region';

++ Goal 'How do I...?' ['Have you tried...?'];

++ Goal ...

++ Goal ...
+++ Hint ..

++ Goal

+ HintMenu 'The Ship'

++ Goal
....

```

This is probably best done in a source file of its own - say `hints.t` - which you keep separate from the rest of your game code. You may need to add a few things to the game proper for your hints system to pick up on, but otherwise your hints system should not change anything in the game proper, and certainly nothing in `hints.t` (or whatever you choose to call it) should change anything in the game state at all. Apart from the availability of hints the game should compile and play just the same whether you include `hints.t` in the build or not.

What the hint system aims to do is to provide a successive list of hints for each of the various goals that the player might be pursuing at any particular point in the game, but not to list any goals that have not yet become relevant or have ceased to be relevant. These goals may optionally be organized in submenus under the main hints menu to aid navigation through the hint system for the player. How this all works in detail, we shall now explore, though it must be emphasized from the outset that the aim here is merely to give enough examples for you to see how the hint system works, not to try to provide a complete set of hints for the game we have created, which would be far too large a task to complete here.

22.2. TopHintMenu

`TopHintMenu` is a class which defines the top of your hints menu tree. You use it to create an object in which to nest the rest of your hint system, and the library will automatically register it as the root of your hint menus.

In the file where you're going to create your hint system, simply define:

```
TopHintMenu 'Hints';
```

22.3. HintMenu

Unless your hint system is going to be very simple, you'll probably want to split it into submenus. To do this you use a series of `HintMenu` objects, which would be located directly in the `TopHintMenu`. In the present game we might do this by splitting the hints according to region:

```
TopHintMenu;
```

TADS 3 Tour Guide

```
+ HintMenu 'In the First Set of Caves'
;

+ HintMenu 'Aboard the ship'
;

+ HintMenu 'The Tardis and its Destinations'
;

+ HintMenu 'On the East Side of the Lake'
;

+ HintMenu 'On the South Side of the Lake'
;

+ HintMenu 'On the West Side of the Lake'
;
```

Note that the Menu template defines the single-quoted string as the title property, so, for example, our first HintMenu could have been defined as:

```
+ HintMenu
    title = 'In the First Set of Caves'
;
```

Note also that each of these menus is only displayed if it contains currently open goals.

22.4. Goal

Goals are the main building blocks of your hint system. They comprise an objective the player is trying to establish, together with a list of hints to help the player towards that goal.

The first thing to appreciate about Goals is that they may be in one of three states: **Undiscovered**, **Open** or **Closed**. A Goal is in the *Undiscovered* state before the player has got to the point where s/he knows that it might be an objective s/he needs to pursue. It is *Open* once the player is (or should be) aware that this is something s/he may need to achieve, and may require hints on it. It becomes *Closed* once the player has achieved this particular objective (and so no longer needs any hints for it). The hints relating to a Goal are offered to the player only when the Goal is Open. While the Goal is undiscovered, displaying even the name of the goal may be giving away information prematurely, and once the Goal is fulfilled, continuing to display it is unnecessary.

Obviously it is up to you, the author, to define when a Goal changes from being *Undiscovered* to being *Open*, and from being *Open* to being *Closed*. But the Goal class defines a number of properties to help you do this:

- **closeWhen** - Determine if there's any condition that should close this goal. We'll check closeWhenSeen, closeWhenDescribed, and all of the other closeWhenXxx conditions; if any of these return true, then we'll return true. See OpenWhen.
- **closeWhenAchieved** - An optional Achievement object that closes this goal. Once the achievement is completed, this goal's state will automatically be set to Closed. This makes it easy to associate the goal with a puzzle: once the puzzle is solved, there's no need to show hints for the goal any more.
- **closeWhenDescribed** - close the goal when the given object is described (by EXAMINE).
- **closeWhenKnown** - an optional Topic or Thing that closes this goal when known
- **closeWhenRevealed** - an optional <.reveal> tag that closes this goal when revealed
- **closeWhenSeen** - An option object that, when seen by the player character, closes this goal. Many goals will be things like "how do I find the X?", in which case it's nice to close the goal when the X is found. See openWhenSeen.
- **closeWhenTrue** - an optional general-purpose check that closes the goal
- **openWhen** - Determine if there's any condition that should open this goal. This checks openWhenSeen, openWhenDescribed, and all of the other openWhenXxx conditions; if any of these return true, then it returns true. Note that this should generally NOT be overridden in individual instances; normally, instances would define openWhenTrue instead. However, some games might find that they use the same special condition over and over in many goals, often enough to warrant adding a new openWhenXxx property to Goal. In these cases, you can use 'modify Goal' to override openWhen to add the new condition: simply define openWhen as (inherited ||

TADS 3 Tour Guide

newCondition), where 'newCondition' is the new special condition you want to add.

- **openWhenAchieved** - An optional Achievement object that opens this goal. This goal will be opened automatically once the goal is achieved, if the goal was previously undiscovered. This makes it easy to set up a hint topic that becomes available after a particular puzzle is solved, which is useful when a new puzzle only becomes known to the player after a gating puzzle has been solved.
- **openWhenDescribed** - this is like openWhenSeen, but opens the topic when the given object is described (with EXAMINE) .
- **openWhenKnown** - An optional Topic or Thing that opens this goal when the object becomes "known" to the player character. This will open the goal as soon as gPlayerChar.knowsAbout(openWhenKnown) returns true. This makes it easy to open a goal as soon as the player comes across some information in the game.
- **openWhenRevealed** - An optional <.reveal> tag name that opens this goal. If this is set to a non-nil string, we'll automatically open this goal when the tag has been revealed via <.reveal> (or gReveal()).
- **openWhenSeen** - An optional object that, when seen by the player character, opens this goal. It's often convenient to declare a goal open as soon as the player enters a particular area or has encountered a particular object. For such cases, simply set this property to the room or object that opens the goal, and the goal will automatically be marked as Open the next time the player asks for a hint after seeing the referenced object. Note that this may not always work as expected, since there may be some forms of discovery (e.g. where an action by the player causes an object to be moved into scope via moveInto and a custom report) that do not result in marking the object as seen, so that openWhenSeen (and closedWhenSeen) may not become true as expected.
- **openWhenTrue** An optional arbitrary check that opens the goal. If this returns true, we'll open the goal. This check is made in addition to the other checks (openWhenSeen, openWhenDescribed, etc). This can be used for any custom check that doesn't fit into one of the standard openWhenXxx properties.

To illustrate the use of openWhen and closedWhen, we'll modify Goal to add our custom openWhenMoved and closeWhenMoved conditions. These can be useful alternatives to openWhenSeen and closeWhenSeen in situations where the library may not mark an object as seen, but either the act of moving it into scope or that of the player character taking it can be relied upon to set moved = true.

```
modify Goal
  openWhenMoved = nil
  closeWhenMoved = nil
  openWhen = (inherited || (openWhenMoved != nil && openWhenMoved.moved))

  closeWhen = (inherited || (closeWhenMoved != nil && closeWhenMoved.moved))
;
```

Next we should list the other properties of Goal:

- **title** - The topic question associated with the goal. The hint system shows a list of the topics for the goals that are currently open, so that the player can decide what area they want help on.
- **goalState** - This goal's current state. We'll start off undiscovered. When a goal should be open from the very start of the game, this should be overridden and set to OpenGoal.
- **isActiveInMenu** - we're active in our parent menu if our goal state is Open
- **location** - The goal's parent menu - this is usually a HintMenu object. In very simple hint systems, this could simply be a top-level hint menu container; more typically, the hint system will be structured into a menu tree that organizes the hint topics into several different submenus, for easier navigation.
- **menuContents** - The list of hints for this topic. This should be ordered from most general to most specific; we offer the hints in the order they appear in this list, so the earlier hints should give away as little as possible, while the later hints should get progressively closer to just outright giving away the answer. Each entry in the list can be a simple (single-quoted) string, or it can be a Hint object. In most cases, a string will do. A Hint object is only needed when displaying the hint has some side effect, such as opening a new Goal.

Normally the only properties you will need to worry about when constructing your hints are *title*, *menuContents* and the various *OpenWhenXXX* and *CloseWhenXXX* conditions. Since the first two are common to all Goal objects, they are defined on the Goal template, so that:

```
+ Goal
  title = 'How do I open the door?'
  menuContents =
  [
    'First find the key. ',
    'Then try unlocking the door with the key. ',
    'Now open the door. '
  ]
;
```

TADS 3 Tour Guide

Can be written simply as:

```
+ Goal 'How do I open the door?'
  [
    'First find the key. ',
    'Then try unlocking the door with the key. ',
    'Now open the door. '
  ]
;
```

(The [Goal template](#) has a couple of extra optional elements, but we'll try to keep things simple here).

After all these preliminaries, we can at last proceed to give a few examples. The first even faintly puzzling obstacle the player is likely to encounter is the large boulder preventing egress west from the main cave. It may be a good idea to provide two sets of hints for this, one pointing the player towards finding the dynamite, and the second prompting the player how to make good use of the dynamite once it's found. The boulder problem will become apparent as soon as the boulder is seen. We want to close one Goal and move on to the next once the dynamite is found, but this won't necessarily work with open/closedWhenSeen since the dynamite isn't necessarily marked as seen when the player discovers it. Instead we'll use our custom open/closedWhenMoved to do the job. CloseWhenMoved will also work nicely to close the second goal, since once the boulder's blown up it's moved into nil, but it cannot be moved by any other means:

```
+ HintMenu 'In the First Set of Caves'
```

```
;
```

```
++ Goal 'How do I get past the boulder in the main cave?'
  [
    'Well, you won\'t be able to push it. ',
    'You\'ll need to find some way of making it disappear. ',
    'Try blowing it up. ',
    'You\'ll need some dynamite. ',
    'There\'s some dynamite not far away. ',
    'But it\'s buried. ',
    'Seen a spade anywhere? '
  ]
  openWhenSeen = boulder
  closeWhenMoved = dynamite
;
```

```
++ Goal 'How do I get rid of that boulder in the main cave?'
  [
    'What did you find in the small sandy cave at the end of the secret passage? ',
    'What might you use dynamite for? ',
    'Are there any fire sources to hand near the boulder? ',
    'Do you want to be holding the dynamite when it detonates? ',
    'Examine the boulder closely. ',
    'Does examining the boulder suggest where you might put the dynamite?',
    'Light the dynamite from the torch on the wall, put in the boulder,
    then scarper until you hear the explosion. '
  ]
  openWhenMoved = dynamite
  closeWhenMoved = boulder
;
```

A slightly trickier Goal to deal with is that for opening the trunk. On the one hand, at what point does the player become aware that unlocking the trunk might be a problem? It's hardly necessary to provide a hint telling the player to go and find the key as soon as s/he sees the trunk. On the other hand, once the key's been found and tried and it fails to open the trunk, the player will recognize that the problem is more complicated than it seemed. This might be a good point at which to open the Goal. But how can the Goal tell that this point has been reached? The neatest way might be to stick a <.reveal> tag in the message that reports that the key fits the lock but won't turn and test for that in the openWhenRevealed property.

On the other hand, when should this goal be closed? The tempting thing would be to test for the trunk being open or unlocked, but this *might* not work, since the player *could* close and lock the trunk again, and if the next time the library checked (i.e. the next time the player asked for hints) the trunk was re-closed or re-locked, the Goal would not be closed. But once the player has opened the trunk s/he will see its contents, so we can test for one of the items inside the trunk being seen:

TADS 3 Tour Guide

```
++ Goal 'How do I unlock the trunk? '
[
  'Well, you\'ve already found the right key. ',
  'You\'ll have to do something to make the key work. ',
  'It will take a long journey to find what you need. ',
  'How might you gain access to future technology?'
]
openWhenRevealed = 'trunk-lock'
closeWhenSeen = glassJar
;
```

The list of hints here is only partial, and would ideally need to be expanded, but instead we'll conclude with a more urgent task, namely ensuring that the appropriate `<.reveal>` tag actually gets revealed when it's meant to:

```
trunk : KeyedContainer, Heavy 'large black trunk' 'large black trunk' @mainCave
  initSpecialDesc = "A large black trunk rests in the middle of the cave. "
  initiallyLocked = true
  keyList = [brassKey]
  lockOrUnlockAction(lock)
  {
    if(gIobj.isBent)
      reportFailure('{The iobj/he} fits the lock but won\'t quite turn in it.<.reveal trunk-lock> ');
    else
      inherited(lock);
  }
;
```

22.5. Hint

As we have seen, most of the hints given within a `Goal` object are single-quoted strings, but there may be occasions when we want a hint to do a bit more than display a message, and in that case we can use a Hint object.

Hint defines two properties and one method:

- **hintText** - the text that is displayed for this hint.
- **referencedGoals** - A list of other Goal objects that this hint references. By default, when this hint is shown for the first time, each goal in this list is promoted from Undiscovered to Open. Sometimes, it's necessary to solve one puzzle before another can be solved. In these cases, some hints for the first puzzle (which depends on the second), especially the later, more specific hints, might need to refer to the other puzzle. This would make the player aware of the other puzzle even if they weren't already. In such cases, it's a good idea to make sure that we make hints for the other puzzle available immediately, since otherwise the player might be confused by the absence of hints about it.
- **getItemText ()** - Get the hint text. By default, we mark as Open any goals listed in our referencedGoals list, then return our hintText string. Individual Hint objects can override this as desired to apply any additional side effects.

Prior to TADS 3.0.9 there was no Hint template, so you would need to define your own if you wanted one (which would be useful since `hintText` and `referencedGoals` are likely to be used a good deal on Hint objects). TADS 3.0.9 now defines:

```
Hint template 'hintText' [referencedGoals]? ;
```

If you keep all your hints in one source file, which is probably a good idea, then this definition need only be placed once, at the start of your hints file, since there's no need for you to be defining Hint objects anywhere else.

The normal way we'd use a Hint object is when the text of a hint suggests another goal that the player might want to pursue, so that this second goal should become an open one. For example, if we provide a series of hints to the player about how to open the door south from anotherCave, we'll sooner or later end up telling the player to put a goat in the slot. This then alerts the player that s/he needs to find a goat, which could then become another goal we want to open it at this point. This is how we can handle this with a couple more Goals linked by a suitable Hint:

TADS 3 Tour Guide

```
++ Goal 'How do I open the south door in Another Cave?'
[
  'Have you examined the door closely?',
  'Have you looked at the slot in the door?',
  'Is there anything written by the slot?',
  groatHint
]
openWhenSeen = lakeDoor
closeWhenTrue = (!lakeDoor.isLocked) // there's no way of relocking this door so this is safe
;

/* Note, there's no need to locate a Hint item inside the Goal that references it,
 * it is simply convenient to do so (a) to show the relationship between the two
 * and (b) so we can carry on adding Goals after the Hint without the containment
 * tree getting messed up.
 */

+++ groatHint : Hint
  'You need to put a groat in the slot. '
  [groatGoal]
;

++ groatGoal : Goal
  'What\'s a groat and where can I find one? '
  [
    'A groat is a small coin. ',
    'More specifically, it\'s a small brass coin. ',
    'You won\'t find it till you\'ve dealt with the boulder. ',
    'How closely have you examined everything in the cave beyond the boulder? ',
    'What might be under the carpet, and how might you find out? ',
    'Have you tried pulling the carpet? '
  ]
  closeWhenDescribed = brassCoin
;
```

We use `closeWhenDescribed` on the `groatGoal` goal, because the brass coin is first named as a 'small brassy object', and does not reveal itself as a coin - specifically a groat - until it has been examined (which sets `described` to true). We do not need to define an `OpenWhenXXX` property on `groatGoal` since `groatCall` is opened by `groatHint`.

23. Further Information

23.1. Concluding Remarks

That concludes *The Quest of the Golden Banana* as far as we are going to take it; it also concludes our tour of the adv3 library.

One thing I can be quite sure of at this point is that I won't have covered some point or other that you hoped to see discussed. This was, after all, a guided tour of the TADS 3 library, not an exhaustive discussion of all its features. The attempt to be fully exhaustive would probably have meant that this Guide would have grown to four times its present size - too large to be genuinely useful - or, even more probably, would never have seen the light of day at all. To have attempted to discuss every possible tweak and modification to the standard library to produce every kind of effect that any game author might want would have been totally impossible. If following this Tour Guide has helped you to feel more confident that you know your way round the main thoroughfares of the adv3 library, it will have served its purpose; the exploration of the side-streets, by-ways and back-alleys is now up to you (with the aid of the comments in the library source code, an adventuring spirit, and plenty of experimentation).

There are, however, some things that have not been covered in this Tour Guide because they are already covered elsewhere, and it is not the purpose of this Guide to regurgitate material that is already reasonably well documented. Instead, a few pointers to that other documentation will be given in the sections that follow.

If there is something really central you feel I have omitted, by all means drop me a line (eric dot eve at hmc dot ox dot ac dot uk) to let me know. If you spot any typos or other errors in this guide, or find any bugs in the Golden Banana game, *please, please* do drop me a line to let me know, so I can correct it for future versions. All feedback and suggestions will be warmly welcomed.

23.2. Language Information

This Guide has made no systematic attempt to describe the TADS 3 language. Most of the language features of TADS 3 are well documented in the *System Manual* that comes with the standard distribution of the TADS 3 Author's Kit. If for any reason you do not have this (perhaps because you are not using the Windows version of TADS 3) you can download it from <http://www.tads.org>. There is also an introduction to the TADS 3 language in Chapter One of *Getting Started in TADS 3*.

23.3. Defining Verbs

Although this Guide has given a number of examples of defining verbs in TADS 3, no attempt has been made to treat this topic in a systematic fashion.

For more information on defining your own verbs in TADS 3, consult the sections in the *Technical Manual* on 'How to Create Verbs', along with articles on 'Action Results' and 'Good use of `verify()` and `check()`'. See also the discussion on 'Controlling the Action' in Section 4 of Chapter Five of *Getting Started in TADS 3*, and on defining `CrossAction` in Chapter Six of *Getting Started*.

23.4. Message Substitution Parameters

Several times during the course of this Tour Guide we have used examples of message substitution parameters (in strings like '{You/he} open{s} {the dobj/him}'). The basic idea behind such strings is that the library replaces them with the appropriate actor and object names at run time. For example, according to context this example might be displayed as "You open the trunk" or "Sarah opens the door". It's accordingly a highly flexible mechanism for writing general-purpose messages that can be automatically customised to a wide variety of particular situations. For a more detailed explanation of how these work, see the article on 'Message Parameter Substitutions' in the *Technical Manual*.

23.5. Past Tense

Since version 3.0.9 TADS 3 has incorporated Michel Nizette's past tense extension, which allows an IF work to be narrated in the past tense instead of the more present tense, either for the complete work, or for certain sections of it (such as flashbacks). To switch a game from present tense to past tense narration (e.g. "On the table was a banana" instead of "On the table is a banana"), you simply need to set `gameMain.usePastTense = true`. To experiment with the effect you could try setting `gameMain.usePastTense = true` on *The Quest of the Golden Banana*; but if you do, you'll probably find that it only half works.

Setting `gameMain.usePastTense` to true will ensure that all the default library messages come out in the past tense rather than the present, but you will also be writing a large amount of text of your own, including room and object descriptions, custom messages, responses to actions and the like. If your whole game is in the past tense, this is no problem, because you can simply write all your custom text in the past tense. If you want your game to switch tenses part-way through, however, you need to write messages that switch between present and past tense depending on the setting of `gameMain.usePastTense`. The comments in `en_us.t` give some details of how to do that, but for more reader-friendly documentation take a look at the section on using the past tense in the *Technical Manual*.

24. Templates

24.1. Achievement Template

An achievement defines its descriptive text. It can also optionally define the number of points it awards.

```
Achievement template +points? "desc";
```

24.2. Actor Template

For actors, we generally override the npcDesc or pcDesc rather than the base desc. For the standard templates, set the npcDesc, since most actors are NPC's rather than the player character.

```
Actor template 'vocabWords' 'name' @location? "npcDesc";
```

24.3. AltTopic Template

Alternative topics just specify the response string or strings.

```
AltTopic template "topicResponse" | [eventList];
```

```
AltTopic template [firstEvents] [eventList];
```

24.4. ConvNode Template

A conversation node needs a name.

```
ConvNode template 'name';
```

24.5. DeadEndConnectorTemplate

A DeadEndConnector specifies a notional destination name and a description of travel via itself.

```
DeadEndConnector template 'apparentDestName'? "travelDesc";
```

24.6. DefaultTopic Template

Default topics just specify the response text .

```
DefaultTopic template "topicResponse" | [eventList];
```

```
DefaultTopic template [firstEvents] [eventList];
```

24.7. Enterable Template

For enterables, allow special syntax to point to the connector which an actor uses to traverse into the enterable.

```
Enterable template ->connector inherited;
```

In this context inherited refers to the [Thing Template](#)

The definition is thus equivalent to

```
Enterable template ->connector 'vocabWords' 'name' @location? "desc";
```

24.8. Exitable Template

```
Exitable template ->connector inherited;
```

In this context inherited refers to the [Thing Template](#)

The definition is thus equivalent to

```
Exitable template ->connector 'vocabWords' 'name' @location? "desc";
```

24.9. EventList Template

An event list takes a list of strings, objects, and/or functions.

```
EventList template [eventList];
```

24.10. Footnote Template

A template for footnotes - all we usually need to define in a footnote is its descriptive text, so this makes it easy to define one.

```
Footnote template "desc";
```

24.11. Goal Template

Template for defining menu items for the hint system

```
Goal template ->closeWhenAchieved? 'title' 'heading'? [menuContents];
```

24.12. Hint Template

Template for defining individual Hints on the Hint system:

```
Hint template 'hintText' [referencedGoals]?
```

24.13. MenuItem Template

```
MenuItem template 'title' 'heading'?
```

24.14. MenuLongTopicItem Template

```
MenuLongTopicItem template 'title' 'heading'? 'menuContents';
```

24.15. MenuTopicItem Template

```
MenuTopicItem template 'title' 'heading'? [menuContents];
```

24.16. MiscTopic Template

Miscellaneous topics just specify the response text or list.

```
MiscTopic template "topicResponse" | [eventList];
```

```
MiscTopic template [firstEvents] [eventList];
```

24.17. MultiLoc Template

Template for multi-location objects. To put a MultiLoc object in several initial locations, simply use a template giving the list of locations.

```
MultiLoc template [locationList];
```

24.18. NoTravelMessage Template

```
NoTravelMessage template "travelDesc";
```

24.19. OneWayRoomConnector Template

For one-way room connectors, provide special syntax to point to the destination room.

```
OneWayRoomConnector template ->destination;
```

24.20. Passage Template

For passages, allow special syntax to point to the master side of the passage.

```
Passage template ->masterObject inherited;
```

Here, 'inherited' refers to the Thing or VocabObject template, so this equates to:

```
Passage template -> masterObject;  
Passage template -> masterObject 'vocabWords';  
Passage template -> masterObject 'vocabWords' 'name' @location? "desc"?
```

24.21. Room Template

For rooms, we normally have no vocabulary words, but we do have a name and description, and optionally a "destination name" to use to describe connectors from adjoining rooms.

```
Room template 'roomName' 'destName'? 'name'? "desc"?
```

24.22. ShuffledEventList Template

A shuffled event list with two lists - the first list is the sequential initial list, fired in the exact order specified; and the second is the random list, with the events that occur in shuffled order after we exhaust the initial list.

```
ShuffledEventList template [firstEvents] [eventList];
```

24.23. SpecialTopic Template

A [SpecialTopic](#) takes a keyword list or a regular expression instead of the regular match criteria. It also takes a suggestion name string and the normal response text. There's no need for a score in a special topic, since these are unique.

```
SpecialTopic template
    'name'
    [keywordList] | 'matchPat'
    "topicResponse" | [eventList] ?;
```

A ShuffledEventList version of the above:

```
SpecialTopic template
    'name'
    [keywordList] | 'matchPat'
    [firstEvents]
    [eventList];
```

24.24. StyleTag Template

```
StyleTag template 'tagName' 'openText'? 'closeText'?
```

24.25. SyncEventList Template

```
SyncEventList template ->masterObject inherited;
```

24.26. Thing Template

```
Thing template 'vocabWords' 'name' @location? "desc"?
```

24.27. ThingState Template

```
ThingState template 'listName_' @listingOrder?;
```

24.28. TopicEntry Template

A TopicEntry can be defined with an optional score, followed by the match criteria (which can be either a single matching object, a list of matching objects, or a regular expression pattern string), followed by the optional response text (which can be given either as a double-quoted string or as a list of single-quoted strings to use as an EventList).

```
TopicEntry template
+matchScore?
@matchObj | [matchObj] | 'matchPattern'
"topicResponse" | [eventList] ?;
```

A ShuffledEventList version of the above:

```
TopicEntry template
+matchScore?
@matchObj | [matchObj] | 'matchPattern'
[firstEvents] [eventList];
```

We can also include *both* the match object/list *and* pattern:

```
TopicEntry template
+matchScore?
@matchObj | [matchObj]
'matchPattern'
"topicResponse" | [eventList] ?;
```

A ShuffledEventList version of the above

```
TopicEntry template
+matchScore?
@matchObj | [matchObj] 'matchPattern'
[firstEvents] [eventList];
```

24.29. TopicGroup Template

A TopicGroup can specify its score adjustment:

```
TopicGroup template +matchScoreAdjustment;
```

24.30. TravelMessage Template

For TravelMessage connectors, provide special syntax to specify the message and point to the destination.

```
TravelMessage template ->destination "travelDesc";
```

24.31. Unthing Template

For an Unthing we normally want to define the notHereMsg property rather than the desc property; along with trying to do anything else to or with an Unthing, examining an Unthing displays its notHereMsg, not its description.

```
Unthing template 'vocabWords' 'name' @location? 'notHereMsg';;
```

24.32. VocabObject Template

```
VocabObject template 'vocabWords';
```

25. Changes

25.1. Changes for v3.0.12

Added a note emphasizing that [Lockable](#), [LockableWithKey](#) and [IndirectLockable](#) need to precede Thing-derived classes in the superclass list of an object declaration.

Amended the [DeadEndConnector](#) template in accordance with its change in TADs 3.0.12.

Added a brief discussion of the new IFID field on versionInfo.

Added a short section on the new ActorByeTopic class.

Removed the sections on the inputManager and the mainOutputStream, since these substance of these has been moved to a new article on Some Common Input/Output Issues in the *Technical Manual*.

25.2. Changes for v3.0.10

- Added a [switch](#) to the Tardis console to open and close the door.
- Changed the implementation of the [back of the small picture](#).
- Added a section on [BagOfHolding](#).
- Removed the random element from the mainDeck connector in [Floorless](#).
- Added information on isSticky, limitSuggestions and blockEndConv for [ConvNode](#).
- Updated the information on the locating of [ConversationNodes](#).
- Modified the definition of the [scales](#) object to use the library property putInName.
- Added further advice on the use of keywordList in [SpecialTopic](#).
- Added an explanation of [setSuperclassList](#) (and setSuperclassList).
- Added a section on [DeadEndConnector](#), and modified appropriate parts of the Banana source code to use it; made corresponding changes to the explanation of [FakeConnector](#).
- Added a section on [OneTimePromptDaemon](#).
- Added a chapter on [ModuleExecObjects](#) (such as [InitObject](#) and [PreinitObject](#))
- Added brief sections on [LeaveByeTopic](#) and [BoredByeTopic](#).
- Explained the addition of an initiallyLocked property to [Lockable](#).
- Changed the discussion of [CollectiveGroup](#) to reflect the new collectiveGroups property.
- Added a brief note on RestrictedHolder and its new subclasses to the end of the section on [RestrictedContainer](#).
- Added an explanation of getEnteredVerbPhrase() to the [PathPassage](#) section.

25.3. Changes for v3.0.9

- Correction to the code for the vending machine slot in [PlugAttachable](#) to prevent the coin reappearing once the ticket is issued.
- Definition of the endGame function in [cannotGoThatWayInDark](#) corrected, and the redundant redefinition in [Dynamite](#) removed.
- Added a section on the [TravelConnector](#) class.
- Added a section on the [mainOutputStream](#) object.
- Changed definition of [longCaveWords](#) to reflect the fact that it's no longer necessary to override isListedInRoomPart().
- [remoteInitSpecialDesc\(\)](#) now takes actor instead of pov as its parameter.
- Use the new macro gTopicText in [DefaultAskTellTopic](#) in place of gTopic.getTopicText.
- Provided examples of dobjMsg and failCheck() which also correct previous bugs on the [glassJar](#) object. The failCheck() method has also been used in one or two other places instead of the previous lengthier construct.
- Corrected an error on the [oilLamp](#) object by using the new failCheck routine.
- Added a note of the new [Hint](#), [SyncEventList](#) and [Unthing](#) templates. The use of the new [Unthing](#) template is also discussed in the Unthing section, and of the Hint template in the [Hint](#) section.

TADS 3 Tour Guide

- Added a discussion of some of the main properties and methods on [Lockable](#), including the new lockStatusReportable.
 - Added a brief discussion of [BasicDoor](#) and noted that [Door](#) and [SecretDoor](#) now inherit from it.
 - Added sections on [Openable](#) and [BasicOpenable](#).
 - Removed iobjFor(BurnWith) code on [RedCandle](#) that's no longer necessary with TADS 3.0.9.
 - Added a brief description of the new moveIntoAdd() and moveOutOf methods for [RoomParts](#).
 - Updated the explanation of verboseMode on [gameMain](#).
 - Added an example of [RoomPartItem](#).
 - Added an explanation of the new SensoryEmanation property isEmanating in the [Odor](#) section.
 - Added a discussion of the revised occludeObj() method and the new isOccludedBy() method to the [Occluder](#) section.
 - Added a brief section on [RandomFiringScript](#) and amended the superclass lists of [RandomEventList](#) and [ShuffledEventList](#) to show that they now inherit from it.
 - Added an example of the new [AutoClosingDoor](#) property reportAutoClose().
 - Added a brief section on using the [past tense](#) in TADS 3.0.9.
- Corrected sundry typos.

25.4. Changes for v3.0.8

The ShipboardRoom class renamed [Shipboard](#).

The custom TardisRoom class is no longer used in the sample game, since it is identical to the new [ShipboardRoom](#).

Added a slightly fuller explanation (or rather, warning against abuse of) the locationList property of [MultiLoc](#).

Added to the explanation of [roomParts](#), specifically in relation to changing or moving them dynamically during the course of a game.

Added an explanation of the new specialTopicHistory object to the description of [SpecialTopic](#).

Changed the [gameMain](#) definition to use the new setAboutBox() method.

Added a brief explanation of the new [gameMain](#) property allVerbsAllowAll.

Amended the final couple of paragraphs in the description of [ComplexContainer](#) to take account of new features (specifically the remapping of isOpen, isLocked, makeOpen and makeLocked to the subContainer).

Amended the definition of the custom [RedCandle](#) class to fix an obscure bug in the sample game.

Added a new section on [Occluder](#).

Corrected a number of typos.

25.5. Changes for July-Sept 2004

Added a 'Traps for the Unwary' Section to the end of the [ComplexContainer](#) chapter.

Added a brief section on the [inputManager](#).

Corrected a number of typos.

25.6. Changes for v3.0.7

Add 'small' to the vocabWords of brassKey (see [KeyedContainer](#)).

Add a line of code to prevent the game becoming unwinnable if the Tardis is parked outside the caves before the rockfall has been triggered by climbing the ladder (see the definition of tardisButton in [DynamicLocations](#)).

TADS 3 Tour Guide

Add an example of an [AskTellShowTopic](#).

Add an example of an [AskTellGiveShowTopic](#).

25.7. Changes for v.3.0.6q

The only change in 3.0.6q affecting this Tour Guide is the change in name from G_Dict to cmdDict, which affects the discussion of [vocabWords](#), although the change is also carried through when the same code for `brassCoin.changeName()` is repeated subsequently

25.8. Changes for May 2004

Alter erroneous references to `startRoom` in the [SecretDoor](#) section to refer to `mainCave`.

25.9. Changes for v3.0.6p

- In [InitiateTopic](#) remove the modification to `ActorTopicDatabase.initiateTopic()`, which has now been incorporated into the library.
- Change to [RoomTemplate](#), which also affects the introductory discussion of [Templates](#) and the discussion of [OutdoorRoom](#).
- Change from `initDesc` to `initSpecialDesc` and `initExamineDesc` to `initDesc` throughout (see especially [initDesc & initSpecialDesc](#)).
- Change of name of `cannotAttachMsg` to `explainCannotAttachTo` on [Attachable](#).
- Discussion and example of the new `deferToEntry` method in relation to [DefaultGiveTopic](#).
- Add `dest` to the argument list of `monolith.beforePushTravel` in [TravelPushable](#).
- The use of `remoteInitSpecialDesc` instead of `distantInitDesc` in the [DistanceConnector](#) section.
- Add example of `inRoomName(pov)` to the [DistanceConnector](#) section.
- Removal of `snowmobile.out=nil` in the [Vehicle](#) section (now incorporated in the library).
- Removal of bug-fix to `AltTopic` discussed in relation to [HelloTopic](#) (now incorporated in the library)
- Alteration in [ByeTopic](#) to reflect the fact that `ByeTopic` now handles both explicit and implicit conversation termination as standard.

25.10. March/April 2004

The following changes were made in March/April 2004

- The definition of the `longCaveWords` in the [Decoration](#) section.
- The change of property name from `roomDesc` to `inRoomDesc` in the [specialDesc](#) section.
- The correction of a typo that referred to `Wearable` instead of `Food` in the [Food](#) section.
- The addition of `inherited()` to `TardisDestination.construct()` in the [DynamicLocations](#) section.
- Substantial changes to the monolith in the [TravelPushable](#) section.
- Some changes to the definition of `redGlow` in the [Vaporous](#) section.
- Some minor alterations to the `vocabWords` of some of the exhibits in the [CollectiveGroup](#) (static) section.

INDEX

#		
#endif	29	
#ifdef	29	
*		
* (plural marker)	55	
—		
__DEBUG	29	
+		
+ syntax	23	
<		
<.convnode>	274	
<.convstay>	274	
<.reveal key>	232, 235	
<.topics>	267	
A		
abandonLocation	85	
AccompanyingInTravelState	219	
AccompanyingState	218	
accompanyTravel	218	
Achievement	307	
action	320	
actionAllowsAll	15	
Actor	212	
Actor Customization	214	
ActorByeTopic	252	
actorDirectlyInRoom	149	
actorInPrep	181	
actorKnowsDestination	21	
ActorState	216	
actorTravelingWithin	179	
addToAgenda	285	
addToScore	113, 306, 307	
addToScoreOnce	307, 310	
addWord	63	
adjective	63	
advanceState	210	
affinityFor	94	
afterAction	47, 77, 216	
afterTravel	216	
AgendaItem	285	
agendaList	285	
allowedPostures	180	
allowPutBehind	87	
allowPutUnder	86	
allowYouMeMixing	15	
allVerbsAllowAll	15	
AltTopic	235	
alwaysListOnMove	85	
aName	66	
anonymous functions	19	
anonymous object	23	
apparentDestName	21	
arrivingTurn	216	
asExit	23	
aslobjFor	77	
AskAboutForTopic	248	
AskConnector	42	
AskForTopic	245	
AskTellGiveShowTopic	249	
AskTellShowTopic	249	
AskTellTopic	244	
AskTopic	240	
atmosphereList	19	
Attachable	170	
attachedObjects	170, 177	
attentionSpan	222	
AutoClosingDoor	32	
autoSuggest	216	
autoUnlockOnOpen	95	
awardPoints	310	
awardPointsOnce	307, 311	
B		
BagOfHolding	94	
banana	9	
baseCannotDetachMsg	177	
BasicChair	180	
BasicContainer	75	
BasicDoor	31	
BasicOpenable	101	
Bed	183	
beforeAction	47, 216, 245	
beforeTravel	176, 216	
bibliographical metadata	17	
blockEndConv	275	
Booth	187	
BoredByeTopic	252	
bottomRoom	38	
break	57	
brightness	27, 29, 76, 102	
BrightnessOff	102	
BrightnessOn	102	
buildLocationList	196	
bulk	70, 77	
bulkCapacity	70, 74, 75, 77, 85, 179	
BulkLimiter	74	
Button	39, 97, 116, 119	
ByeTopic	225, 251	
C		
callWithSenseContext	108, 165, 236	
canAttachTo	170, 176	
canBeSensed	113	
canBeTouchedBy	108	
canDetachFrom	170	
Candle	103	
canEndConversation	275	
cannotAttachMsg	174	
cannotDetachMsg	170	
cannotGoThatWay	50	
cannotGoThatWayInDark	51	
cannotGoThatWayMsg	50	
cannotKissActorMsg	214	
cannotMoveMsg	59	
cannotPushMsg	59	
cannotTakeMsg	55, 59, 60, 149	
cannotUnlockMsg	96	

TADS 3 Tour Guide

canObjReachContents	185
canObjReachSelf	185
canPushedObjectPass	151
canPutIn	81
canReachFromInside	185
canReachSelfFromInside	185
canReturnItem	84
canTravelerPass	22, 34, 42, 151, 191
Chair	183
check	320
checkPreCondition	189
checkTravelBarriers	42, 43
Closed	315
closeWhen	315
closeWhenAchieved	315
closeWhenDescribed	315
closeWhenKnown	315
closeWhenRevealed	315
closeWhenSeen	315
closeWhenTrue	315
CollectiveGroup	199, 202
collectiveGroups	199
CommandTopic	293
ComplexComponent	90
ComplexContainer	90, 93
Component	60, 118
connector	23, 41
connectorBack	43
connectorMaterial	164
connectorStagingLocation	43
connectorTravelPreCond	43
construct	119, 310
Constructor	119
Consultable	301
ConsultTopic	303
Container	76
ContainerDoor	92
Containers	74
ConvAgendaItem	287
Conversation Nodes	274
ConversationReadyState	223
ConvNode	275
ConvType	253
createUnlistedProxy	43
curScriptState	205, 250
curSetting	117
curState	216
CustomFixture	55
CustomImmovable	60
CyclicEventList	207

D

Daemon	137
DarkRoom	29
darkTravel	43
DeadEndConnector	21
debugging commands	29, 123
fiat lux	29
force open	123
Decoration	55
DefaultAskForTopic	264
DefaultAskTellTopic	259
DefaultAskTopic	258
DefaultCommandTopic	294, 299

DefaultConsultTopic	304
defaultFloor	111
replacing	111
DefaultGiveShowTopic	263
DefaultGiveTopic	260
defaultPosture	180
DefaultShowTopic	262
DefaultTellTopic	259
DefaultTopic	257
deferToEntry	260
Defining Verbs	320
DelayedAgendaItem	288
delegated	170
desc	62, 307
describeArrival	43
described	69
describeDeparture	43
describeMovePushable	149
descWithoutSource	160, 162
descWithSource	160
destination	23, 28
destName	19, 24, 149
DigWith	111
dir	50
direction	50
DirectObject	90
disambigName	72
discover	113
discovered	113
Dispensable	84
Dispenser	84
displaySchedule	160, 162
DistanceConnector	153
Distant	57
Distinguishing command phrasings	34
dobjFor	32
dobjMsg	81
Door	30
doScript	26, 205, 210
dyanmic object creation	84
Dynamic Locations	119
Dynamite	108, 135

E

emanationHereDesc	160
endConvBoredom	275
endConvBye	275
endConversation	252, 275
endConvTravel	275
endEmanation	160
endGame	51
Enterable	23, 36
enteringRoom	52
EntryPortal	23, 41
escortActor	220
escortDest	220
escortStateClass	220
eventList	26
EventList	205
eventManager	135
eventPercent	209
eventReduceAfter	209
eventReduceTo	209
eventualLocation	114

TADS 3 Tour Guide

examineStatus	100
excludeMatch	260
execAfterMe	144
execBeforeMe	144
execute	119
exit	47, 79
Exitable	23, 36
exitDestination	179
ExitOnlyPassage	32
ExitPortal	23
explainCannotAttachTo	170
explainTravelBarrier	34, 42, 43, 191
ExternalEventList	210

F

failCheck	81
FakeConnector	20
feelDesc	72
fiat lux	29
finishGameMsg	51, 108
FireSource	103
firstEvents	210
fixedSource	43
Fixture	55, 75
Flashlight	103
Floor	48
Floorless	39
FloorlessRoom	38, 48
Food	72
force open	123
fuelLevel	103, 105, 108
function	77
Fuse	135

G

gActionIn	47
gActionIs	47
gadgets	116
gameMain	15
gDobj	79
getAccompanyingTravelState	218, 219
getActor	216
getApparentDestination	43
getBestMatch	245
getBulk	85
getDestination	43
getEnteredVerbPhrase	34
getFacets	36, 58
getItemText	318
getNextValue	214
getOrigText	34
getScriptState	205
getState	26
getSuperclassList	70
getTopicText	264
getWeight	77, 123
GiveShowTopic	232
GiveTopic	229
glass	76
globalParamName	66, 212
gMessageParams	137, 151
Goal	315
goalState	315
Greeting Protocols	225

gRevealed	232, 235
gSetKnown	215
GuidedInTravelState	222
GuidedTourState	220

H

handleAttach	170
handleDetach	170
handleTopic	226, 229, 262
hasSeen	215
Heavy	60
HelloGoodbyeTopic	253
HelloTopic	225, 250
hereWithoutSource	160, 162
hereWithSource	160
HermitActorState	218
Hidden	113
HiddenDoor	39, 59
HighNestedRoom	184
Hint	318
HintMenu	314
hintText	318

I

IFID	17
Immovable	59
impByeTopicObj	251
ImpHelloTopic	251
impliesGreeting	250
InConversationState	222
inConvState	223
IndirectLockable	96, 176
inherited	9, 48
initDesc	54, 59, 65, 189
initializeVocabWith	63
initialLocationClass	196
initiallyActive	285
initiallyLocked	79, 95, 96
initiallyOpen	101
initiallyPresent	114
initiateConversation	274, 279
InitiateTopic	236
initNominalRoomPartLocation	55
InitObject	142, 144
initSpecialDesc	65
inRoomDesc	66
inRoomName	53, 153
instanceObject	196
Intangible	153
Intangibles	153
intiallyLocked	97
invokeItem	285
iobjMsg	81
isActive	226, 235, 240
isActiveInMenu	315
isAmbient	158, 160
isAttachedTo	170
isCircularPassage	43
isCollectiveAction	199, 202
isConnectorApparent	43
isConnectorListed	43
isConnectorPassable	43
isConnectorVisibleInDark	43
isConversational	226

TADS 3 Tour Guide

isDirectlyIn	137
isDone	285
isEmanating	160
isEquivalent	84, 103
isHeldBy	102
isHer	212
isHim	212
isIn	137
isInitiallyIn	196
isInitState	216
isListed	54, 112
isListedInContents	54, 84, 112
isListedInInventory	54
isListedInRoomPart	55
isLit	102, 103
isLocked	95
isMajorItemFor	170
isMasked	160, 162
isMatchPossible	226
isMyKey	100, 133
isOccludedBy	155
isOn	129
isOpen	76, 101
isPermanentlyAttachedTo	170
isPlural	77
isProperName	212
isPulled	123
isReady	285, 287, 288
isSticky	275
isValidSetting	117, 118, 127
isWorn	73
itIt	212

K

Key	97
KeyedContainer	97, 133
keyIsPlausible	133
keyList	97
Keyring	100, 133
keywordList	277
Knowledge	215
knownProp	215
knowsAbout	215

L

LabeledDial	117
Language Information	320
LeaveByeTopic	252
leavingRoom	52
Lever	123, 131
libMessages	311
libScore	312
light	29
LightSource	102
limitSuggestions	273, 275
listing strings	127
location	315
locationList	164, 195, 196
Lockable	95
LockableContainer	79
LockableWithKey	99
lockedDesc	95
lockOrUnlockAction	97
lockStatusObvious	95

lockStatusReportable	95, 100
logicalRank	189
lookAroundWithinName	149
lookInVaporousMsg	157
LookupTable	119

M

makeLit	102, 135
makeLocked	79, 95, 96
makeOn	129
makeOpen	27, 101
makePresent	114
makePresentByKey	114
makePresentByKeyIf	114
makePresentIf	114
makeProper	212
makePulled	123, 131
makeSetting	117, 118, 127
mast	38
masterObject	25, 30, 210
Matchbook	107
matchObj	226
matchPattern	226
matchScore	226
matchScoreAdjustment	256
Matchstick	107
matchTopic	226, 229, 240, 293
material	76
maxEntries	277
maxPoints	307
maxScore	312
maxSetting	119
maxSingleBulk	70, 74
maybeRemapTo	131
menuContents	315
Message Properties	157
Message Substitution Parameters	320
min	77
minBulk	85
minSetting	119
MiscTopic	253
mix-in class	95
modify	103
modify VerbRule	103
moved	59, 65, 114
moveInto	65, 196
moveIntoAdd	48, 196
moveIntoForTravel	216
moveOutOf	48, 196
moveWhileAttached	170
Moving Actors Around	216
MultiFaceted	197
MultiFacetedFacet	197
MultiInstance	196
MultiInstanceInstance	196
MultiLoc	38, 55, 158, 195
myDispenser	84
myItemClass	84

N

name	62, 267
NameAsOther	79
NearbyAttachable	174
NestedRoom	179

TADS 3 Tour Guide

new	84, 135, 137, 140, 142
Daemon	137
Fuse	135
SenseDaemon	142
SenseFuse	140
new function	26
nextState	222
Noise	162
noLongerHere	160
NominalPlatform	181
NonPortable	54
noResponse	218
noteActive	275
noteLeaving	275
noteTraversal	20, 25, 43
notHereMsg	58
notifyInsert	77, 96, 176
notifyProp	165
notifyRemove	77, 84
notifySightEvent	236
notifySoundEvent	165, 236
notImportantMsg	55
NoTopic	277
NoTravelMessage	31, 32
notWithIntangibleMsg	157
noun	63
NPC Knowledge	215
npcContinueList	275
npcContinueMsg	275
npcGreetingList	275
npcGreetingMsg	275
NumberedDial	119

O

obeyCommand	216, 292, 295
objNotWorn	76
obviousPostures	180
occludeObj	155
Occluder	155
Odor	160
OilLamp	105
OneTimePromptDaemon	142
OneWayRoomConnector	34, 36
OnOffControl	129
opaque	164
Open	315
Openable	100
OpenableContainer	77
openDesc	101
openStatus	90, 100
openWhen	315
openWhenAchieved	315
openWhenDescribed	315
openWhenKnown	315
openWhenRevealed	315
openWhenSeen	315
openWhenTrue	315
otherSide	23
OutdoorRoom	19, 48
OutOfReach	185
Overriding obeyCommand	295

P

parameter substitution	47
------------------------------	----

Past Tense	321
PathPassage	34
perInstance	310
PermabentAttachmentChild	177
PermanentAttachment	177
Person	212
Platform	180
playerActionMessages	81
plKey	114
PlugAttachable	176
plural	63
plurals	55
points	307
PostRestoreObject	146
PostUndoObject	147
posture	181
PraiseTopic	253
preCond	76
PreCondition	189, 202
preconditions	320
preinitialization	145
PreinitObject	119, 145, 260, 311
PreRestartObject	147
PreSaveObject	146
PresentLater	114, 130
PromptDaemon	142, 164
promptMessage	42
PushTravelBarrier	151
putInName	77

Q

quest	9
-------------	---

R

rand	19, 262
RandomEventList	209
RandomFiringScript	211
Readable	71, 75
readDesc	71
readyTime	288
RearContainer	87
RearSurface	89
referencedGoals	318
remapTo	28, 32, 90, 133
rememberTravel	43
remoteInitSpecialDesc	153
remoteSpecialDesc	153
removeEvent	135, 137, 142
removeMatchingEvents	135, 137
removeWord	63
reportAutoClose()	32
reportFailure	79, 81
resetItem	285
RestrictedContainer	81, 96, 130
revealHiddenItems	74
Room	24, 48
roomAfterAction	47
RoomAutoConnector	30
roomBeforeAction	47
RoomConnector	22
roomDaemon	19, 236
roomDesc	66
roomName	19, 179
RoomPart	48

TADS 3 Tour Guide

RoomPartItem	55
roomParts	48, 55, 111, 123
replacing defaultFloor	111
Rooms and Connectors	18

S

sample game	9
saving a game	146
sayBurnedOut	103, 108
sayDeparting	219
scoreCount	307
scoreRankTable	311
Script	19, 205
scriptedTravelTo	216
SecretDoor	27, 123
SecretFixture	79, 119, 155
seenProp	215
sense	165
SenseConnector	153, 164
SenseDaemon	142
SenseFuse	140
SensoryEmanation	153
SensoryEvent	165
setConvNode	274
setCurState	216
setDelay	288
setHasSeen	215
setKnowsAbout	215
setSuperclassList	70
Settable	118
setToInvalidMsg	118
Shipboard	36
ShipboardRoom	36, 99
ships	36
showExitsInStatusLine	15
ShowTopic	230
showTravelDesc	20
ShuffledEventList	210
ShuffledList	214
ShuffledTextList	19
shuffleFirst	210
SightEvent	236
sightPresence	112
sightSize	153
SimpleAchievement	310
SimpleNoise	160
SimpleOdor	158
SingleContainer	93
smellDesc	72, 158
SoundEvent	165, 236
SoundObserver	165
soundSize	164
sourceDesc	160
SpaceOverlay	85
specialDesc36, 54, 59, 66, 140, 149, 189, 216, 219, 330	
specialDescOrder	140
specialNominalRoomPartLocation	55
SpecialTopic	277
specialTopicHistory	277
SpringLever	117
stagingLocations	184
StairwayDown	25, 38
StairwayUp	25, 32, 38

Startup Code	15
stateAfterEscort	220
stateDesc	216
static	48
StopEventList	26, 207
StretchyContainer	85
stringLister	127
subContainer	90, 92
subLocation	90
subRear	90
subSurface	90
subUnderside	90
SuggestedAskTopic	269
SuggestedGiveTopic	271
SuggestedNoTopic	271
SuggestedShowTopic	271
SuggestedTellTopic	270
SuggestedTopic	267
SuggestedTopicTree	271
SuggestedYesTopic	271
suggestTopics	267
Surface	75
Switch	130
switch statement	57
SyncEventList	210

T

takeTurn	216
Tardis	99
initial definition	99
targetObj	79
tasteDesc	72
TCommandTopic	296
TellTopic	243
template	318, 322, 323, 324, 325, 326, 327
Achievement	322
Actor	322
AltTopic	322
ConvNode	322
DefaultTopic	322
defining	318
Enterable	323
EventList	323
Exitable	323
Footnote	323
Goal	323
Hint	323
MenuItem	323
MenuLongTopicItem	324
MenuTopicItem	324
MiscTopic	324
MultiLoc	324
OneWayRoomConnector	324
Passage	324
Room	325
ShuffledEventList	325
SpecialTopic	325
StyleTag	325
SyncEventList	325
Thing	325
ThingState	326
TopicEntry	326
TopicGroup	326
TravelMessage	326

TADS 3 Tour Guide

VocabObject	327
Templates	9
tense	
past	321
Thing	62
ThingMatchTopic	226
ThroughPassage	28, 32
throwTargetCatch	207
timesToSuggest	267, 270
title	314, 315
toInteger	119
tooFullMsg	74
TopHintMenu	314
Topic	240
TopicEntry	226
TopicGroup	256
TopicMatchTopic	226
topicResponse	226
totalPoints	307
TourGuide	220
transient	147
transparent	164
transSensingThru	164
travelAction	42
travelBarrier	42, 43, 151
TravelBarrier	42, 191
TravelConnector	43
travelDesc	20, 31
travelerArrving	52
travelerPreCond	189
TravelMessage	29
travelObjs	42
travelObjsPhrase	42
TravelPushable	149
travelTo	216
TravelVia	32
travelWhileAttached	170
TravelWithDesc	20
TravelWithMessage	26, 32

triggerEvent	165
tryImplicitAction	176

U

Underside	86
Undiscovered	315
UntakeableActor	212
Unthing	58
useInitDesc	66
uselessToAttackMsg	214
usePastTense	15, 321
useSpecialDesc	66, 140, 189

V

validContents	81, 96
validSettings	117
valueList	214
Vaporous	157
Vehicle	189
VehicleBarrier	191
verboseMode	15
VerbRule	103
modify	103
verbs	320
defining	320
verify	320
vocabWords	63

W

weak token	63
Wearable	73
weight	70, 77
weightCapacity	70
wornBy	73

Y

YesTopic	277
----------------	-----