Learning TADS 3

by Eric Eve

(for version 3.0.18)

Table of Contents

1 Introduction

1.1 The Aim and Purpose of this Manual

TADS 3 is an extremely powerful and versatile system for writing well-polished works of Interactive Fiction, but at first sight it can look quite overwhelming, since there seems to be so much to learn. TADS 3 in fact makes many common IF coding tasks extremely easy, but there *is* a lot to learn, and it's very hard to produce anything worthwhile with it without first mastering the basics.

Several years ago, when I was still fairly new to TADS 3 and had just been struggling to get to grips with it on the basis of the extremely limited documentation then available, I conceived the idea of producing a guide for people like myself who had some familiarity with Inform 6 or TADS 2 but wanted to get up to speed on TADS 3. The result was *Getting Started in TADS 3*. When Mike Roberts subsequently asked if I would be happy for *Getting Started* to be included in the official TADS 3 documentation set (for the official launch of TADS 3 at version 3.0.12) I was happy to agree, and also spent some time trying to adapt *Getting Started* to its new role. Both before and since then *Getting Started in TADS 3* has proved helpful to many people, but it clearly hasn't suited everyone. The present manual was conceived as an alternative to *Getting Started* for people to whom *Getting Started*'s approach is uncongenial or unhelpful.

While *Learning TADS 3* is still a tutorial, unlike *Getting Started* it is not tied to taking the reader through the development of a sample game, and is thus free to present the material in a far more systematic manner. It will therefore suit readers who would prefer a more systematic treatment, or who don't feel they will learn much by copying the code for someone else's game. On the other hand *Getting Started* may work better for readers who would like to be taken through the development of a complete game, or who would like more step-by-step guidance. If you're anxious to get on with development your own IF masterpiece, *Learning TADS 3* may be the better choice for you (and below I'll offer a couple of suggestions on how you might like to use it in tandem with developing your own game). If what you're after is a gentler walkthrough of TADS 3's capabilities, then you may prefer *Getting Started*.

The writer of this kind of manual is inevitably in a no-win situation. Reader A will complain loudly that points X, Y and Z haven't been covered. Reader B will complain about being overwhelmed by too much information. Reader A and Reader B may very well be the same person at different times! This manual attempts to steer a middle course between the complaints of these two readers by setting out what all TADS 3 users need to know, offering pointers to what many TADS 3 users will want to know, and largely ignoring features that are likely to be used only occasionally. It also attempts to avoid excessive repetition of material available in other parts of the TADS

3 documentation set, although a considerable amount of overlap has turned out to be unavoidable. Since we all start from different places and want to write different kinds of games, there is no one-size-fits-all ideal manual that will perfectly suit everybody.

In order to master (or even become moderately competent at) a complicated system like TADS 3 (and TADS undoubtedly *is* complicated) it's necessary to learn both about the library (in particular the various classes that can be used to define objects in a TADS 3 game) and the language (in particular how to go about various coding tasks). To be sure, some people may want to find all about the language first, but such people can read the *TADS 3 System Manual*. For the benefit of everyone else, this manual intersperses information about coding (mainly from the *System Manual)* among the seemingly more 'practical' sections on how to use the library. That way readers can gradually acquire what they need to know about the language in the course of learning about and trying out the more concrete aspects of game writing.

Becoming proficient at TADS 3 is not a matter of committing everything one might possibly want to know about it to memory. The system is far too large for that (though it is not quite so bewilderingly vast as it is sometimes made out to be). Becoming proficient at TADS 3 is a matter of learning (through practice) those parts of the system that one uses most commonly, while at the same time learning to use the documentation effectively in order to look up the rest. Since effective use of the *TADS 3 System Manual*, the *Library Reference Manual* and the *TADS 3 Technical Manual* are essential skills for any TADS 3 author, readers of this book will be encouraged to look material up in these other documents from an early stage. There would, in any case, be little or no point in reproducing large amounts of information that are perfectly well covered elsewhere, and so throughout this book readers are referred to these other manuals for further information. The aim is to present the basic information here and to leave readers to find out the less common details elsewhere. This should benefit readers in two ways: first, by helping them to become familiar with other parts of the TADS 3 documentation, and second, by allowing the explanations offered here to be kept relatively simple, concentrating on what is basic and central.

How readers choose to use this book is, of course, up to them. If anyone really wants to use it as the libretto for *TADS 3 – The Opera* I can hardly prevent them! But perhaps I may be permitted to offer a suggestion. Some readers may find it helpful (perhaps after trying out an exercise or two from the first chapter) to read fairly rapidly through the whole book without stopping to do the exercises or to look up the suggested material in other manuals, and only then come back to work through this book more slowly and carefully second time round, trying out all the exercises and looking up all the other suggested material. This approach could have a number of benefits: by enabling you to satisfy your natural curiosity about what is coming next in the first read-through, it should help you to curb the urge to race through too quickly, skimping on the exercises and external cross-references, when you come to read through it second time round. It should also give you an initial overview so that when you come to read this book through more carefully second time round it will be with at least some idea of how the parts fit into the whole.

Again there's more than one way you can use the exercises. You can, if you wish, take them literally and try to implement exactly what they suggest (or where they refer to sample games you could try playing the sample games and then recreating them in your own code). Or, where the exercises suggest comparison with a sample game, you could just give a bit of thought as to how you might implement the suggested game, maybe jotting a few notes, and then study the source code of the sample game. Or, if you're primarily anxious to get on with your own game, you could try to think of something in your own game that's analogous to the exercise being proposed and then go ahead and implement that, perhaps studying the source code of the sample game when one's suggested to get ideas for your own game. That way, you will be able to make progress with your own game while still following this manual in a reasonably systematic fashion.

1.2 What You Need to Know Before You Start

It is assumed that anyone reading this book has a reasonable idea of what Interactive Fiction is, how it's played, and what its basic conventions are, otherwise they wouldn't be wanting to learn how to program in TADS 3. It's also assumed that you know the absolute basics of compiling a game in TADS 3. If not, you should first read Parts I and II of the *TADS 3 System Manual*. If you are using a Windows system you should also start Windows Workbench and read the documentation that comes with that. And even if you've chosen to use this book instead of *Getting Started in TADS 3* you might find it useful to read the section on 'Creating Your First Project' in Chapter 1 of *Getting Started* and possibly Chapter 2 as well.

At several points in this book you'll be invited to try writing your own (short) games. When you start a new game you'll need a source file that contains (at a minimum) the following:

```
#charset "us-ascii"
#include <adv3.h>
#include <en_us.h>

versionInfo: GameID
    name = 'My Practice Game'
    byLine = 'by an Aspiring Author'
    version = '1'
;

gameMain: GameMainDef
   initialPlayerCharacter = me
;
me: Actor
   location = startRoom
;
```

If you don't call your starting location startRoom, then you should change startRoom to the name of the room where you want your game to start (you may also want to change the name of the game and your own name in the byLine).

If you’re using Workbench, you can just use the File -> NewProject option (from the menu) and then ask to create an “advanced” game in order to start with the skeletal code shown above. If you’re not using Workbench you can still copy the file starta3.t from the ..\samples directory of your TADS 3 author’s kit installation; copy starta3.t to your working directory, rename it to something else (mygame.t, for example) and you can use it as the basis for your own exercises.

Finally, many of the exercises contained in this manual refer to sample games you can look at. A complete set of these sample games can be obtained from http://www.tads.org/learning_tads3_sample_games.htm.

1.3 Feedback and Acknowledgements

Special thanks are due to Jim Aikin, Mark Engelberg and Knight Errant for pointing out various errors in an earlier draft of this manual and making various suggestions for improvements.

If anyone else wishes to point out errors or offer suggestions, I can be contacted on eric.eve@hmc.ox.ac.uk.

2 Map-Making – Rooms

2.1 Rooms

No game can take place without a room, so the very first thing we have to learn to define is precisely that – a room. Let's suppose our game starts in a bedroom, so that this is the room we want to define. It might look something like this:

```
bedroom: Room
      roomName = 'Bedroom'
      desc = "Your bed lurks in one corner, the clothes a heap from a
      restless night. The only way out is to the east. "
;
```

Note that we have defined two properties of the bedroom object: roomName (what the room is called) and desc (its description). These two properties are so common that we can define a room without stipulating them explicitly, by means of what's known as a *template*. A template is simply a convenience feature of TADS 3 that lets us define commonly-used properties without explicitly stating which properties we're defining; this works by defining these common properties in a particular order (sometimes with additional symbols like + or @ or -> to identify parts of the template). The Room template is a very straightforward one; using the room template our room definition becomes:

```
bedroom: Room 'Bedroom'
      "Your bed lurks in one corner, the clothes a heap from a
      restless night. The only way out is to the east. "
;
```

We've mentioned a way out to the east, so presumably that must go somewhere. Let's suppose is goes out to the landing. To allow movement between rooms we can define the `east` property of the bedroom to point to the landing, then define a new room, the landing, with its `west` property pointing back to the bedroom.

```
bedroom: Room 'Bedroom'
      "Your bed lurks in one corner, the clothes a heap from a
      restless night. The only way out is to the east. "
      east = landing
;

landing: Room 'Landing'
      "The landing runs from west to east; your bedroom lies west. "
      west = bedroom
;
```

If you compiled this game, it wouldn't be terribly exciting, but it would at least let you move backwards and forwards (or rather east and west) between the two rooms.

When we wrote `east = landing` and `west = bedroom`, we were adding properties to the room which describe where the player character goes when the player attempts to move in the corresponding direction. The standard directional properties available in TADS 3 are the eight compass directions: **north**, **south**, **east**, **west**, **northeast**, **northwest**, **southeast**, **southwest**, together with the four special directions: **up**, **down**, **in** and **out**. In Shipboard rooms (see below) we can also use the directions **port**, **starboard**, **fore** and **aft**.

Exercise 1: See if you can create a larger map, using the tools we have seen so far. Traditionally people start by creating a map of their own home or place of work, and you could do that. But by all means feel free to use your own imagination if you'd like to try something more adventurous.

2.2 Coding Excursus 1: Defining Objects

The rooms we have been creating are examples of *objects*. A great deal of programming in TADS 3 consists in defining objects. A typical object definition in TADS 3 looks something like:

```
objectName: ClassList
  property1 = 'some text'
  property2 = somethingElse
;
```

The *objectName* is simply a name we give to the object so that we can identify it elsewhere in our code (for example we used the name `landing` which enable us to attach the `landing` to the `east` property of the `bedroom`).

The *ClassList* is a list of one or more *classes* to which the object we're defining belongs. A *class* defines the kind of object it is; objects belonging to the same class generally have quite a bit of behaviour in common, so we use classes to define that common behaviour. So far the only class we've met is `Room`, but TADS 3 has many other classes we can use, and we can also define our own. We'll soon be meeting some more.

Each object (and class) can have a number of *properties*. These are pieces of data associated with the object, for example the `name` and `desc` of a Room. These pieces of data can be of many kinds, such as strings (pieces of text), numbers, lists, and other objects. Objects (and classes) can also have *methods*, but we'll talk about them later.

In the example above the object definition is terminated with a semicolon (;). An alternative form of object definition uses braces, like this:

```
objectName: ClassList
{
  property1 = 'some text'
  property2 = somethingElse
}
```

Some kinds of object have properties we use so often that there's a short-cut method of defining them, using what TADS 3 calls a *template*. The only template we've met so far is that for the **Room** class, which defines a short-cut means of defining the **roomName** and **desc** properties.

Templates can be a great time-saver when defining objects, but how do we know which properties they define? One way is to look them up in the *Library Reference Manual.* This is a tool all TADS 3 authors need to get to grips with sooner or later, so we may as well start now.

Open the *Library Reference Manual* from the TADS 3 Bookshelf (it's best if you can open it in a web browser and keep it available all the time). Along the top of the *LRM* you should see a row of links looking like:

Intro	*Classes*	*Actions*	*Grammar*	*Objects*	*Functions*	*Macros*	*Enums*	*Templates*	*all symbols*

Click on the *Templates* link near the right hand end. The bottom left-hand frame of the *LRM* should then change to a list headed with the title Templates. Scroll down the list till you find *Room*, and then click on the *Room* link. A definition of the Room template should then appear near the top of the main frame, looking like this:

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

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.

Anything followed by a question-mark is an optional part of the template. This tells us that when we define a room with a template, the item in single-quote marks will be the **roomName** property. If there's a second item in single-quote marks it will be the **destName**, and if there's a third it will be the **name** (don't worry about what these mean for the moment). Whether we define the Room with one, two, or three items in single-quoted strings, the item that appears between double-quotes will be the **desc** property.

So, for example, if we defined:

```
auditorium: Room 'Auditorium of Albert Hall' 'the auditorium'
    "The auditorium is thronged with a great press of people. "
;
```

Then 'Auditorium of Albert Hall' would the **roomName**, 'the auditorium' would be the **destName**, and "The auditorium is thronged with a great press of people. " would be the **desc**. We'll explain **destName** further below.

After a while, the common templates (e.g. for **Room**) quickly become familiar, and we don't need to look them up any more. Some beginners find templates a little confusing, however, since until they become familiar it isn't always clear what properties they're defining. For that reason we'll be careful to introduce each template as we first use it, and to give at least one example of each new class where where the properties are all defined explicitly. You may also find it helpful to download the template quick-reference chart from http://www.tads.org/t3dl/TemplatesQref.zip.

For a fuller explanation of the material covered in this Coding Excursus, see the chapter on "Object Definitions" in Part III of the *TADS 3 System Manual*. Note, however, that some of the material in that chapter relates to concepts we shall be meeting in later Coding Excursuses.

2.3 Different Kinds of Room

So far we've only met one kind of room, defined with the `Room` class. There are in fact a number of classes of Room we can use, depending on the kind of room we want:

- `Room`: This kind of room comes with four walls, a ceiling and a floor as standard. It's the class we'd typically use to define an indoor location, a room in the ordinary sense of the word.
- `OutdoorRoom`: This is like a `Room`, except that it has no walls, and comes with ground and sky instead of floor and ceiling. It's the kind of `Room` we'd typically use to define an outdoor location.
- `ShipboardRoom`: This is just like a `Room`, except that we can define the directions `port`, `starboard`, `fore` and `aft` here. We might typically use this for cabins aboard a ship.
- `DarkRoom`: This a room with no light, so the player won't be able to see anything in it unless a light source is brought it. We could use this kind of room for a darkened cellar or an underground cave, for instance.
- `FloorlessRoom`: This is a room with no floor (although it still has four walls and a ceiling).

Although this covers several possibilities, it clearly doesn't cover them all. It might be dark outdoors or aboard ship for example, or we might be out on deck rather than in a a cabin (we want shipboard directions but not a ceiling and four walls). If we have a location representing the top of a tree, it might well be floorless, but `FloorlessRoom` won't cover it, because when we're at the top of the tree we wouldn't have four walls and a ceiling either.

We can create these additional kinds of room using *mix-in* classes with *multiple inheritance.* Multiple inheritance means that we use more than one class in the definition of our object. A mix-in class is one that doesn't stand on its own and has to be mixed in with one or more other classes; whenever we define an object with multiple classes, the mix-in class should always come first.

The mix-in classes we can use with rooms are `Floorless` and `Shipboard`. So, for example, to define our room representing the top of a tree we might define:

```
topOfTree: Floorless, OutdoorRoom 'Top of Tree' 'the top of the tree'
    "The ground looks a long way down from here. A branch stretches out to the
     east, just about wide enough and sturdy enough for you to crawl along. "
    down = footOfTree
    east = branch
    bottomRoom = footOfTree
    cannotGoThatWayMsg = 'Since you can\'t walk on air, that\'s not a
      practicable option. '
;
```

This assumes that `footOfTree` and `branch` are two other rooms we've defined appropriately. Note that we've also introduced a new property here, `bottomRoom`. If we drop something in a room, it will normally fall to the floor, but a floorless room doesn't have a floor, so anything dropped there will carry on falling. The `bottomRoom` property defines where it will end up (in this case something dropped at the top of the tree will fall to the foot of the tree).

Actually, we introduced two new properties. The other one is `cannotGoThatWayMsg`; this is the message that would display if the player tried to go any way but east or down from the top of the tree. Note how we use the backslash (\) immediately before single-quote marks that we want to appear within a single-quoted string property.

Note also that we can use the same template for `OutdoorRoom` as we could for Room. Indeed, we can use this template for all the kinds of room we've seen, because they all inherit from the Room class. We'll explain inheritance in more detail shortly.

We could define a deck aboard ship in the same way:

```
quarterdeck: Shipboard, OutdoorRoom 'Quarterdeck'
   "From here you can see forward into the waist of the ship. "
;
```

But there's no Dark mix-in class we can use to create a dark OutdoorRoom, or a dark version of any of the other kinds of location. For this we have to use yet another property: `brightness`. For a Room the value of this property is normally either 0 (for a dark room) or 3 (for one which is well lit). We can also use a value of 2 for a room where there's enough light to see, but not enough to read by. A brightness of 1 is not meaningful for a room.

So to create a darkened outdoor location, we could define something like the following:

```
graveyard: OutdoorRoom 'Graveyard'
   "This deserted graveyard feels especially spooky by night. The gloomy
    outline of the church casts a deep shadow in the moonlight, from which
    ranks of gravestone emerge as if awaiting some signal to wake to ghostly
    life. "
   brightness = 0
;
```

Exercise 2: Now that we've seen more different kinds of room, try to construct a more adventurous map using examples of each kind, and of the new properties we've just seen. This might, for example, include the inside of a house and part of its garden, or you might try to map out the cabins and decks of a yacht; or you could even try both!

2.4 Coding Excursus 2 – Inheritance

In the previous coding excursus we showed how objects are defined, and noted that each object definition had to contain a list of one or more classes. These are the classes from which the object *inherits*. Classes can also inherit from one or more other classes. For example, `OutdoorRoom` and `DarkRoom` both inherit from `Room`. `FloorlessRoom` inherits from both `Floorless` and `Room`, while `ShipboardRoom` inherits from the two classes `Shipboard` and `Room`. Inheriting from more than one class is called *multiple inheritance*.

Classes are extremely useful when we want to define a several objects (or maybe a whole lot of objects) that basically behave alike. All rooms are basically similar: they can contain actors and other objects; we can move from one room to another using compass directions; when we enter a room we see its room name and its description; all rooms can have varying levels of brightness; using the **look** command in a room works in basically the same way for all rooms; and so on. It would be tedious to have to define all this behaviour for each and every room in our game, but fortunately the `Room` class does it all for us; we can just define our rooms to be of class `Room` and leave the library to do all the rest. The rooms we define in our game all inherit from the `Room` class.

But as we've also seen, there are various kinds of room, each only slightly different from the others. An `OutdoorRoom` is just like a `Room`, except that it has no walls, and has ground and sky in place of floor and ceiling. A `ShipboardRoom` is just like a `Room`, except that shipboard directions (port, starboard, aft and fore) work there. It would a lot of unnecessary work for `OutdoorRoom` and `ShipboardRoom` to define all over from scratch the behaviour they each have in common with `Room`, so they inherit it all from `Room` instead. Then all `OutdoorRoom` and `ShipboardRoom` have to do is to define the ways in which they differ from `Room.`

In fact, `ShipboardRoom` hardly even has to do that. As we've seen, there's a `Shipboard` mix-in class that does the job of making the shipboard directions work, so all we need to do is to mix the two classes together to create the `ShipboardRoom` class:

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

Note the use of the `class` keyword here when we're defining a new class instead of a new object. TADS 3 treats classes and objects in fairly similar ways, but there are differences, so if we mean something to be used as a class, we should define it as a class.

ShipboardRoom also exemplifies the value of multiple inheritance. Where we want to combine the functionality of more than one class, we simply include all the classes we want to inherit from in our class list (whether we're defining an object or a new class). For the most part we can simply define our object as inheriting from multiple classes and leave TADS 3 to work out how all the classes will actually work together, and 99 times out of 100 we'll get the behaviour we expect, provided we observe the one golden rule of multiple inheritance: *mix-in classes must always come first*.

A mix-in class is something like **Floorless** or **Shipboard** that modifies the behaviour of other classes but doesn't define a full set of behaviour itself. A mix-in class is generally any class that doesn't (directly or indirectly) inherit from **Thing**. We'll investigate the **Thing** class in the next chapter.

For more information on object-orientation and inheritance read the article on "Object-Oriented Programming Overview" in the *TADS 3 Technical Manual*, and the chapter on "The Object Inheritance Model" in the Part III of the *TADS 3 Library Reference Manual*.

2.5 Two Other Properties of Rooms

Rooms have quite a few more commonly-used properties beyond those we've mentioned so far. Quite a few of these will be mentioned in later chapters as they become relevant, but there are two we may as well mention here.

Sometimes it's nice to be able to give a room a different description the first time it's examined, perhaps emphasizing the things that first strike the player character's eye or including a reference to how the player character came to be there (something we shouldn't normally do in a room description that could be repeated under other circumstances). For this purpose we can define a **roomFirstDesc** (a double-quoted string), which will be shown the first time the room is described (the **desc** property being used thereafter).

When the player types an **exits** command a list of exits from the current location is displayed; this includes the names of the rooms the exits lead to, provided the player has visited those rooms. This is fine if we've given the room a name like 'graveyard', since we'll see it listed as something like 'east, to the graveyard', but it's not so good if our room name is 'East Street' or 'Hall (west)' since we'll then see a list like 'north, to the east street, or south, to the hall (west)', when it would clearly be better to see a list like 'north, to East Street, or south, to the west end of the hall.' For this purpose we can define a **destName** property (a single-quoted string) with the name we want to be used when the room is shown in a list of exits, for example:

```
hallWest 'Hall (west)'
```

```
    "This large hall continues to the east. "
    destName = 'the west end of the hall'
;
```

The need to define a **destName** that's different from the **roomName** is so common that, as we've seen, we can define it as the second single-quoted string in the Room template:

```
hallWest 'Hall (west)' 'the west end of the hall'
    "This large hall continues to the east. "
    east = hallEast
    roomFirstDesc = "The first thing you notice about this hall is that it
      continues to the east. "
;
```

3 Putting Things on the Map

3.1 The Root of All Things

So far the maps we've created have been pretty dull, since they've consisted purely of empty rooms. In a real work of Interactive Fiction there'd be all sorts of objects in the rooms. Some of them would be portable objects the player can pick up and take from place to place, some would be fixtures like doors, windows, trees, houses and heavy furniture, and some would be mere decorations, objects mentioned in the room description, say, which can be examined but respond to any other kind of command by telling the player that they're not important.

The basic kind of object that's the ancestor of all these kinds of thing is the **Thing**. We use the **Thing** class itself for ordinary objects that the player can pick up and move around. A typical definition of a **Thing** might look like:

```
redBall: Thing
    vocabWords = 'small red hard round cricket ball*balls'
    name = 'red ball'
    location = frontLawn
    desc = "It's quite small and hard; it looks much like a cricket ball. "
;
```

Since these four properties are so commonly used when defining **Things** (virtually every **Thing** is likely to need them), it should come as no surprise that there's a template that can be used when defining **Things**. Using the template, the red ball could be defined like this:

```
redBall: Thing 'small red hard round cricket ball*balls' 'red ball' @frontLawn
    "It's quite small and hard; it looks much like a cricket ball. "
;
```

Study this example very carefully. It applies not only to Thing but to every class that inherits from Thing, which is likely to cover the vast majority of simulation objects defined in any TADS 3 game. If you never get round to learning any other TADS 3 template you should learn this one. You should become so familiar with it that you have no difficulty recognizing at sight which property is which when you see an object defined like this. (It also helps to be just about equally familiar with the **Room** template, especially if you plan on defining quite a number of rooms).

We should now consider each of these common properties in turn.

vocabWords defines the words the player can use to refer to the object in commands. We start by listing all the adjectives the player might use to refer to the object, separating them with spaces. We then list the nouns, separating them with slashes (/). For example, if we thought the player might describe the red ball as a red sphere, we might define its **vocabWords** as 'small red ball/sphere'. Finally we can add a plural noun (or several plural nouns) following an asterisk, hence `'*balls'` at the end of the `vocabWords` of our `redBall` object. The purpose of this is to allow commands like **take balls** or **examine balls** to operate on each and every ball in scope (provided they all have the plural 'balls' in their `vocabWords` property), whereas **take ball** or **examine ball** might cause the parser to ask the player which ball is meant. Note that an object inherits all the adjectives, nouns and plurals defined on the object or class it inherits from *in addition to* any that are defined in its own `vocabWords` property. In other words the `vocabWords` property is *additive.* With most properties, the code you write will replace a property (or method) that is defined on the class, but adding new `vocabWords` to an object doesn't override existing `vocabWords` that are defined on the class.

The `name` property defines the name of the object as it will appear in a room description or inventory listing, e.g. "You see a red ball here" or "You are carrying a red ball."

The `desc` property defines the description of the object that is displayed when the object is examined. Note that unlike the `vocabWords` and `name` properties, which use single quotes, the `desc` property always uses double quotes ("desc").

The `location` property defines where the object is at the start of play. For the time being we'll stick to locating objects in rooms, although later on we'll see other places they can go. Note that the `location` property can *only* be used to define the initial location of an object. *Never* try to move an object by changing its `location` property directly. Call its `moveInto(`*`newloc`*`)` method instead, e.g.:

```
   redBall.moveInto(backLawn);
```

But to talk of methods is to get ahead of ourselves. Instead we'll mention a second very common way of stipulating the initial location of an object. Instead of defining its location explicitly, either through setting `location = wherever` or by using `@wherever` in the template, we can put it after the object it's located in and precede it with a plus sign. For example:

```
frontLawn: OutdoorRoom 'Front Lawn'
   "The front lawn is a relatively small expanse of grass. The somewhat larger
    back lawn lies to the south. "
   south = backLawn
;
+ redBall: Thing 'red round small cricket hard ball*balls' 'red ball'
    "It's quite small and hard; it looks much like a cricket ball. "
;
```

Exercise 3: Add some Things to one of the maps you created earlier. Try running the resulting game; you should be able to pick up these new objects and move them around.

There are a few more things we should know about `vocabWords`. As we've seen, the `vocabWords` property can contain a combination of adjectives, nouns, and plurals. The significance of this is that to match a given Thing, the player can describe it with as many adjectives as s/he likes, but (apart from an exception we shall look at below), only one noun, and the adjectives must comes before the noun. So, for example, if the `vocabWords` property of a given object is 'big red ball/sphere' it can be referred to as 'ball' or 'red sphere' or 'big red ball' but not 'ball sphere'. It can also be referred to by one or more adjectives alone, such as 'big red', but the parser will prefer a noun match to an adjective match. That means, for example, that if we have an orange bucket (with `vocabWords` = 'large orange plastic bucket', say) and an orange (the fruit) with `vocabWords` = 'round juicy orange*fruit', say, then a command like **x orange** will be taken to refer to the fruit rather than the bucket (when both are in scope).

The exception to the rule that the player can't refer to a thing with more than one noun is when the two nouns are separated by 'of'. For example, if we give a pair of shoes object the `vocabWords` 'black pair/shoes' then the player can refer to them as 'pair of shoes' or 'pair of black shoes'.

Incidentally, this illustrates a further two points. If we're going to give something a name that's plural (like 'shoes') we should set its `isPlural` property to true so that the parser will understand what the player means if s/he uses a plural pronoun ('them') to refer to the object, and also so the parser can construct grammatically correct messages about the objects ("the shoes are here" rather than "the shoes is here"). But is an object like 'pair of shoes' singular or plural? If the `name` property of the object is 'pair of shoes' then we should arguably leave it as singular (we'd probably want "the pair of shoes is here" rather than "the pair of shoes are here"), but the player might equally refer to this object as 'it' (thinking of the pair) or as 'them' (thinking of the shoes). In this situation we can also set `canMatchThem` to true, which will let the pair of shoes object match 'them' as well as 'it'. In the reverse case, when we define a plural object (with `isPlural = true`) that the player might refer to as 'it' we can define `canMatchIt = true`.

Normally, an object can match on *any* of the words in its `vocabWords` property. There are occasions when we may not want a match to occur on a particular word when it's used alone. For example we may have so many objects in our game that are described as 'black' or a 'pair' what we don't want the parser to match the black pair of shoes if either or both of the words are used without 'shoes'. To do that we can designate these words as *weak tokens* by putting them in parentheses; for the shoes the `vocabWords` property might then become '(black) (pair)/shoes'. We might also do this with the 'orange' in the orange plastic bucket, just to make absolutely sure the bucket can never be mistaken for the fruit!

On a different matter, in the previous chapter we saw that we can give a Room a `firstRoomDesc` property to describe it the first time it's seen. The analogous property on Thing is `initDesc`. If we give something an `initDesc` property then this will be used to describe it when it's examined until the object has been moved. Or rather, until the object's **isInInitState** property is no longer true (which, by default, is until the object has been moved). We could override this if we liked. For example, when an object is examined its **described** property becomes true. So if we wanted an **initDesc** to be used the first time an object is examined, we could also define **isInInitState = (!described)** on the object. Note, however, that this will also affect when **initSpecialDesc** (which we'll meet a short way below) is used.

3.2 Coding Excursus 3 – Methods and Functions

In discussing how to change the location of a Thing, we introduced a *method*. A method is the other kind of thing you can define on an object besides a property. While a property simply holds a piece of data, a method contains code that's executed when the method is invoked (although, as we shall see, we can generally use a method to provide a value wherever TADS 3 expects a property). A method starts with an open brace **{** and ends with a closing brace **}**. A simple method might look something like this:

```
myObj: object
    name = 'nameless'
    changeName(newName)
    {
         name = newName;
    }
;
```

The method has a single *parameter* called **newName**, which we can use to pass a piece of data to the method. In general a method can take as many parameters as we like (separated by commas), or it can have none at all. The example above is about as simple as a method can get; it simple assigns the value of **newName** to the name property of **myObj**, so that if some other piece of code were to execute the command:

```
   myObj.changeName('magic banana');
```

Then the name property of **myObj** would become 'magic banana'.

There's a few further points to note about this example:

- Every line of code we write (something that's meant to be executed some time) must end with a semi-colon. Note however that this applies only to lines of code in methods and functions, *not* to property declarations and the like.
- To execute a method on a particular object we write the object name, then a dot, then the method name (hence **myObj.changeName('magic banana')**). We'd refer to an object property in the same way (e.g. **myObj.name**).

A method can also return a value to its caller, using the **return** keyword. For example, we might define the (admittedly trivial method):

```
myObj: object
    double(x)
    {
      return 2 * x;
    }
;
```

Then if we executed the statement:

```
   y = myObj.double(2);
```

We'd end up with y being 4.

Sometimes we might want to define some code that we don't want associated with any particular object. In such cases we can use a *function* instead. To define double() as a function we could just do this:

```
double(x)
{
  return 2 * x;
}
```

Then we could just execute statements like:

```
  y = double(x);
```

We'll take a closer look at the kind of statements we can put in methods and functions later. For now, if you want to know more about methods and functions, you can read about them in the Procedural Code chapter of the *TADS 3 System Manual*.

3.3 Some Other Kinds of Thing

We have been introduced to the `Thing` class, which we can use for basic portable objects. But there are also several special kinds of `Thing` – subclasses of `Thing` – which we can use for special purposes. Some of the main examples include:

- `Wearable` – clothing the player character can put on and take off
- `Food` – something the player character can eat.
- `Readable` – something particularly suitable for reading. Normally **read x** behaves the examine as **examine x**, but if **x** is a `Readable` and has a `readDesc` property defined, then **read x** will display the `readDesc` property instead of the normal `desc` property.
- `Flashlight` – a portable light source than can be turned on and off. We'll be looking at light and darkness in more detail later on, but it's helpful to know about this one to provide a means of looking around in dark rooms.

There’s also a couple of special classes we can use to hide things and make them appear at some later point in the game:

- **Hidden** – an object that remains hidden from view until we call its **discover()** method.
- **PresentLater** – an object that starts off the map but comes into play (at the location we define it) when we call its **makePresent()** method. Note that **PresentLater** is a mix-in class (which must therefore be used with some other **Thing** based class.

These last two may become clearer with a couple of examples. Suppose the heroine’s engagement ring has somehow become lost and has somehow got itself under the carpet, so we need to **look under carpet** to find it. We might define the ring as:

```
+ ring: Hidden, Wearable 'diamond silver gleaming engagement ring*rings'
    'diamond ring'
   "It's a silver band with a single gleaming diamond. "
;
```

Then (to anticipate some features of TADS 3 we've yet to meet), we could make the ring appear when the player looks under the carpet:

```
+ carpet: Immovable 'dark red carpet*carpets' 'carpet'
    "It's a plain dark red carpet."

    dobjFor(LookUnder)
    {
        action()
        {
             if(ring.discovered)
                 "You find nothing else under the carpet. ";
             else
             {
                 ring.discover();
                 "You find a ring!";
             }
         }
    }
;
```

If there are parts of this example you don’t understand, don’t worry, they’ll be covered later. The other main thing to note here is that a **Hidden** object has a **discovered** property that tells us whether it has been discovered yet.

We could produce much the same effect using **PresentLater**:

```
+ ring: PresentLater, Wearable 'diamond silver gleaming engagement ring*rings'
    'diamond ring'
   "It's a silver band with a single gleaming diamond. "
;

+ carpet: Immovable 'dark red carpet*carpets' 'carpet'
    "It's a plain dark red carpet."
    dobjFor(LookUnder)
    {
        action()
        {
            if(ring.seen)
                "You find nothing else under the carpet. ";
            else
            {
                ring.makePresent();
                "You find a ring!";
            }
         }
    }
;
```

Although the effect is similar, the mechanism is a little different. The **`Hidden`** ring is present all the time, it's simply invisible until we call its **`discover()`** method. The **`PresentLater`** ring actually starts out off the map and is only moved to the same location as the carpet when we look under the carpet (so that calling **`makePresent()`** on a **`PresentLater`** sets its **`moved`** property to true, which calling **`discover()`** on a **`Hidden`** does not). The **`seen`** property of the ring becomes true once the player character has seen the ring. We'll look at some of the other features of these examples in the next Coding Excursus.

Exercise 4: Try adding some **`Wearable`**, **`Food`** and **`Readable`** objects to your map. Also, add a **`Flashlight`** which can be used to light up a **`DarkRoom`**.

Exercise 5: To work effectively with TADS 3 you need to be able to look things up easily in the *Library Reference Manual*. If you haven't got it open already, open the *LRM* now in your web browser. Click the *Classes* link near the top left hand corner, then scroll down the list of classes in the bottom left-hand panel till you find **`Thing`**. Click on **`Thing`** and take a quick look at its subclass tree; this is the complete list of all the standard TADS 3 classes that derive from **`Thing`**. Don't worry about trying to understand all of them just yet! Instead just spend a bit of time looking further down the page at the properties and methods of **`Thing`**, and then do the same with the other classes we've introduced so far. Don't worry if you can't take it all in – you almost certainly won't be able to; the point is rather to get an initial feel for what's there and for how to use the *Library Reference Manual* to look up the information you need.

3.4 Coding Excursus 4 – Assignments and Conditions

In the previous Coding Excursus we introduced methods and functions, which are the two places procedural code can occur in TADS 3. One of the most common kinds of procedural statement are assignment statements, that is statements that assign a new value to a property or variable.

We've already met properties. Assigning a new value to a property (i.e. changing its existing value to something else within a method or function) is simply a matter of writing the property name, followed by an equals sign (=), followed by the name of the new value we want to assign to the property, for example:

```
   ring.bulk = 2;
   ring.name = 'gold ring';
```

If this code were executed in a method of the ring object, we wouldn't need to specify that it was the ring object's properties we were referring to. In this special case we could just write:

```
   bulk = 2;
   name = 'gold ring';
```

Assignment statement can also perform calculations:

```
   ring.bulk = ring.bulk + 2;
   ring.name = 'gold ' + ring.name;
```

As well as assigning values to properties, we can also assign them to *local variables*. A local variable is simply a temporary storage area for some piece of data. A variable can be local to a method or function, or to some smaller block of code, where a block of code is any sequence of statements between opening and closing braces: `{ }`. A local variable must be declared with the keyword `local` the first time it's used, and the declaration can optionally be combined with an assignment statement, for example:

In the second example, the `+` operator carries out string concatenation. If `ring.name` previously held the value 'ring' then executing the statement `ring.name = 'gold ' + ring.name` will change `ring.name` to 'gold ring'. In the first example the + operator does what you'd expect; it adds 2 to the value of `ring.bulk`. We can also use the other obvious arithmetic operators: - (subtract), * (multiply), and / (divide). For the complete list of operators available in TADS 3 assignment statements, see the section on 'Expressions and Operators' in the *TADS 3 System Manual*. These include some neat short-cuts; for example, `ring.bulk = ring.bulk + 2` can be written as `ring.bulk += 2`.

```
myObj: object
    myMethod(x, y)
    {
        local foo;
        local bar = x + y;
        foo = bar * 2;
        return foo;
    }
;
```

In this method, the parameters **x** and **y** also act much like local variables within the method. They do not have to be declared with the **local** keyword, since they've already been declared as the method's parameters, but like the local variables **foo** and **bar** they are meaningful only within the context of the method.

Method calls, function calls, and assignment statements are probably the most command kind of statements making up the procedural code found in TADS 3 method and functions. Often, both types of statement can occur at once, as in:

```
    foo = bar(x);
```

But there's another kind of statement that's almost just as important, namely *flow-control* statements. Of these probably the most significant is the **if** statement. In programming Interactive Fiction (as in most other kinds of programming), it's seldom enough just to be able to execute a set of statements in set sequence, we often need our code to do different things depending on whether some condition is true or false. The simplest form of an **if** statement in TADS 3 is:

```
   if(condition)
      statement;
```

For example, we might write:

```
     if(ring.weight > 4)
         "The ring feels strangely heavy. ";
```

Which means that if **ring.weight** is greater than 4, the text "The ring feels strangely heavy." will be displayed.

We can optionally add an else clause, which defines what happens when the condition in the **if** part is not true, for example:

```
      if(ring.weight > 4)
          "The ring feels strangely heavy. ";
      else
          "You pick up the ring with ease. ";
```

A further complication is that we might want to execute more than one statement in the **if**-part or the **else**-part. We can do that by enclosing a *block* of statements in braces, thus:

```
      if(ring.weight > 4)
      {
         "The ring feels strangely heavy, so heavy that it the attempt to
          lift it drains your strength. ";
         me.strength -= 3;
      }
      else
      {
          "You pick up the ring with ease. ";
          ring.moveInto(me);
```

```
     }
```

The conditions we can test for include

• `a == b`	a is equal to b
• `a != b`	a is not equal to b
• `a > b`	a is greater than b
• `a < b`	a is less than b
• `a >= b`	a is greater than or equal to b
• `a <= b`	a is less than or equal to b
• `a is in (x, y, z)`	a is equal to x or y or z
• `a not in (x, y, z)`	a is neither x nor y nor z

Note the distinction between `a = b`, which assigns the value of `b` to `a`, and `a == b`, which tests for equality between `a` and `b`.

All these conditional expressions evaluate to one of two values, `true` or `nil`. The `nil` value also has other uses, in contexts where it means roughly 'nothing at all'. A value of `nil` or `0` (the number zero) is treated as false, anything else is treated as true.

It can be useful to combine these logical conditions with *Boolean* operators. The three Boolean operators available in TADS 3 are:

• `a && b`	a and b – true if both a and b are true (i.e. neither nil nor 0)
• `a || b`	a or b – true if either a or b is true (i.e. neither nil nor 0)
• `!a`	not a – true if a is false (i.e. either nil or 0)

Finally, it's often useful to be able to assign one value to a variable or property if some condition is true, and another if it's false, as in:

```
     if(obj.name == 'banana')
          colour = yellow;
     else
          colour = green;
```

This is so common that there's a special conditional operator we can use to write this sort of thing much more succinctly:

```
     colour = obj.name == 'banana' ? yellow : black;
```

More generally, this takes the form:

```
     someValue = condition ? valueIfConditionTrue : valueIfConditionFalse;
```

Together assignment statements, method and function calls, and conditional statement make up the great bulk of procedural statement we're likely to use in TADS

3 programming. There are others, some of which we'll meet later. In the meantime, if you want to get the full picture, read the section on 'Procedural Code' in the *TADS 3 System Manual*.

3.5 Fixtures and Fittings

Objects of class `Thing` are portable: they can picked up, carried around the game map, and dropped elsewhere. This is also true of the various subclasses of `Thing` we met above. But many objects in a work of Interactive Fiction aren't portable, they're part of the fixtures (doors, windows, trees, houses, mountains etc.) or they're too big and heavy to pick up (large tables, sofas, and other actors, for example).

The TADS 3 library defines a `NonPortable` class to cover all these kinds of thing. We don't in fact define any objects to be of class `NonPortable`, however; in a game we'd always use one of `NonPortable`'s many subclasses (which we'll look at shortly). Nonetheless, all objects in these `NonPortable` subclasses have certain characteristics in common:

- They can't be picked up.
- They are not listed in room descriptions (unless they have a `specialDesc` or `initSpecialDesc` property defined).

Note that we *can* define `specialDesc` and/or `initSpecialDesc` on ordinary portable objects too; the `initSpecialDesc` will be displayed until the object has moved, and the `specialDesc` used thereafter (actually the full story is slightly more complex than that, since we can change the condition that takes an object out of its `initState`). For example, we might define:

```
+ ring: Wearable 'diamong ring*rings' 'diamond ring'
    initSpecialDesc = "A diamond ring lies discarded on the ground. "
;
```

This would result in the ring being given a separate paragraph in the room description, and listed as "A diamond ring lies discarded on the ground" until it's moved. The real point here, however, is that a NonPortable objected won't be mentioned in a room listing at all (because it's assumed that it will have been mentioned in the room description) unless it's given a `specialDesc` or `initSpecialDesc`. For example:

+ table: Heavy 'large wooden table*tables furniture' 'large wooden table'

```
      specialDesc = "A large wooden table occupies the middle of the room. "
;
```

Since this table will (probably) never be moved, it doesn't make any difference whether we use `specialDesc` or `initSpecialDesc` in this latter instance.

There are two main classes of `NonPortable` (although these each have subclasses too): `Fixture` and `Immovable`. The distinction between them is quite subtle, and we're not yet ready to say exactly what it is in technical terms yet (since we've yet to meet the distinction between check and verify). But as a first approximation (which will be good enough for most practical purposes), we'd use `Fixture` for things that are obviously fixed in place (like doors, houses, trees, lamp-posts and mountains) and `Immovable` for things that happen to be too large or heavy for the player character to pick up (like large pieces of furniture or other people). The practical difference is that the parser will consider a `Fixture` to be an unlikely target of a command that involves moving (like **take**) *before* deciding what object the player's command refers to, whereas an attempt to move an `Immovable` will simply be disallowed (*after* the parser has decided what object the player's command refers to).

We can (and often will) use the `Fixture` and `Immovable` classes to define objects, but they each have a number of subclasses. We'll start with the subclasses of `Immovable`:

- `CustomImmovable` – This is the same as Immovable, except that we can use one property to define the response to a number of actions (we'll explain this further below).
- `Heavy` – An Immovable that gives being too heavy as the reason why it can't be moved; this is useful for large pieces of furniture and the like.
- `TravelPushable` – An object that can't be picked up, but can be pushed from one room to another (with commands like **push trolley north**).
- `UntakeableActor` – An actor (animate object) that's too large to be picked up (a cow or horse perhaps). For human actors we use `Person`, a subclass of `UntakeableActor`.

The explanation of `CustomImmovable` mentioned properties to provide responses to certain commands. The time has come to introduce some of those properties:

- `cannotTakeMsg –` a message (typically given as a single-quoted string, e.g. 'You can't take that') shown in response to an attempt to take the object in question.
- `cannotMoveMsg` – a message (again typically a single-quoted string) shown in response to attempts to move, drop, push, pull or throw the object in question.
- `cannotPutMsg` – a message (again typically a singe-quoted string) shown in response to put the object in question in, on, under or behind something.

The way these might be used (on either an Immovable or a Fixture) is illustrated by the following example:

```
cabinet: Immovable 'large wooden bulky polished cabinet*furniture cabinets'
    'large wooden cabinet'
    "It's a large piece of furniture, made of polished wood. "
    cannotTakeMsg = 'The cabinet is far too bulky for you to carry around. '
    cannotMoveMsg = 'It is too heavy to move. '
    cannotPutMsg = 'You cannot put the cabinet anywhere else; it is too bulky. '
;
```

The point about a `CustomImmovable` (or, indeed, a `CustomFixture`), is that we can just define the `cannotTakeMsg` property, then the `cannotMoveMsg` and `cannotPutMsg` properties will automatically copy it.

Incidentally, note how we've defined 'furniture' as a plural noun in the `vocabWords` of the cabinet object; there might be several items of furniture in the room, and if we give them all a plural of 'furniture' then the player can refer to them collectively as 'furniture' in commands like **examine furniture**.

Finally, we should list some of the kinds of `Fixture` we can use (there are several others, but we'll be meeting them later):

- `Component -` an object that's part of another object (for example, the handle of a pan or a panel in a door).
- `CustomFixture` – the same as `Fixture` except that we can just define one property, `cannotTakeMsg`, then `cannotMoveMsg` and `cannotPutMsg` will automatically copy it.
- `Decoration` – an object that's unimportant but mentioned in the description of something else, so we want to provide a description of it. If the player attempts to do anything with a Decoration apart from examining it, it will display it's `notImportantMsg`, which is typically 'The whatever is not important. '
- `Distant` – an object representing something that's beyond the player's reach, generally because it's a long way off, like the moon or a distant range of hills. An attempt to do anything but examine a Distant object will result in a refusal of the form 'The moon is too far away. '; this message can be change by overriding the `tooDistantMsg` property.
- `RoomPart` – a special kind of Fixture used to define walls, floor, sky and ceiling. There are two subclasses of `RoomPart`: `DefaultWall` and `Floor`. The library also defines a number of standard `RoomPart` objects: `defaultFloor`, `defaultCeiling`, `defaultNorthWall`, `defaultEastWall`, `defaultSouthWall`, `defaultWestWall` (defined as standard on a `Room`), and `defaultGround` and `defaultSky` (defined as standard on an `OutdoorRoom`). The list of `RoomParts` associated with any Room is defined in that Room's `roomParts` property. This allows us, for example, to define a single `defaultFloor` object that can be used as the floor in every Room.
- `SecretFixture` – a kind of Fixture used for internal implementation only and invisible to the player (so we normally wouldn't give it any `vocabWords`). If you can't immediately think of a use for this class, don't worry.

- **Unthing** – an object used to represent the *absence* of something. This will respond to any command with its **notHereMsg**, typically something like 'The gold ring isn\'t here. '. This might be used, for example, to remind the player that the gold ring has just fallen through a grating. If an **Unthing** and something other than an **Unthing** both match the player's command, the parser will always ignore the **Unthing**.

One last point, because all these objects are ultimately subclasses of Thing, we can (and usually will) use the **Thing** template with them. That means that if we define:

```
table: Heavy 'large round white table*tables' 'large round table'
   "It's a large round table with a white top, resting on a single pedestal. "
;

+ pedestal: Component 'stainless steel pedestal*pedestals' 'pedestal'
    "It looks as if it's made from stainless steel. "
;
```

Then 'large round white table*tables' and 'stainless steel pedestal' would define the **vocabWords** of the two objects, 'large round table' and 'pedestal' would define their **name** properties, and "It's a large round..." and "It looks as if..." would define their **desc** properties. **Unthing**, however, has a special template of its own, since it's generally more useful to define its **notHereMsg** property:

```
ring: Unthing 'gold ring*rings' 'gold ring'
    'The gold ring is no longer here; you dropped it down the grating. '
;
```

In this example the 'gold ring*rings' and 'gold ring' are the **vocabWords** and **name** properties as before, but 'The gold ring is no longer here; you dropped it down the grating. ' defines the **notHereMsg** property (which will be used in response to any command targeted at the ring).

Exercise 6: Look up the **NonPortable** class in the *Library Reference Manual* and take a quick look at what it inherits from and the list of classes that inherit from it (don't worry about the ones we haven't encountered yet). Spend some time exploring its methods and properties and also those of the various subclasses of **NonPortable** we've just been looking at above. Don't worry about anything you don't understand yet, and don't imagine that you have to commit all this information to memory; the point of the exercise is just to get a feel for what's there and to start learning where to find it when you need it.

Exercise 7: Go back to the practice map you created before (or create a new one) and add some examples of each of the various kinds of **NonPortable** object described above (you can skip **RoomParts** and **SecretFixtures**).

4 Doors and Connectors

4.1 Doors

When we're creating a map in a work of Interactive Fiction, we quite often want to create doors. This may simply be for the sake of realism: rooms inside a house or office generally do have doors between them, and it would be a strange house that lacked a front door; or it may be because we want the door to be some kind of barrier, preventing access to some area of the map until the player has obtained the relevant key or solved some other puzzle. We'll deal with doors as barriers in a later step; for now we'll concentrate on doors as connectors between locations.

A physical door has two sides; in TADS 3 the two sides of a door are implemented as separate objects and then linked together. One side of the door is located in the room the door leads from, and the other in the room the door leads to (of course the distinction is a bit arbitrary, since TADS 3 doors, like real doors, work perfectly well whichever way one goes through them). The two sides of the door are then linked by setting the `otherSide` property of one side to point to the other side. In fact, we don't generally set the `otherSide` property directly; we set the `masterObject` property and let the library take care of setting `otherSide` to the value of `masterObject`. The two properties do have two theoretically different purposes: `otherSide` determines where an actor ends up when he or she goes through the door, whereas `masterObject` keeps the two sides of the door in sync when one side of the door is open or closed (or locked or unlocked).

To implement a TADS 3 door we use the `Door` class. This inherits from a number of classes, including `Openable` (which we'll say more about in just a minute), and, via `ThroughPassage` (which we'll say more about a little further below), `Fixture`. Because all Doors are Fixtures, it's not possible for the player to pick them up and carry them away, and they're not listed separately in room descriptions (unless we give them a `specialDesc`); it's generally assumed that we'll mention any relevant doors in the description of the room.

The way to set up a pair of doors should become clearer with an example; suppose the front door of a house leads out from the hall to the drive:

```
hall: Room 'Hall'
     "The front door leads out to the north. "
     north = frontDoor
;
+ frontDoor: Door 'front door*doors' 'front door'
;

drive: OutdoorRoom 'Front Drive'
     "The front door into the house lies to the south. "
     south = frontDoorOutside
;
```

```
+ frontDoorOutside: Door 'front door*doors' 'front door'
    masterObject = frontDoor
;
```

One important thing to notice here is that when we use a door between rooms, we point the relevant compass properties on the rooms in question (e.g. `north` and `south`) to the *door* objects and not to the rooms where the doors lead. Otherwise players would be able to move between rooms without using the doors at all!

Note also that because a `Door` is a kind of `Fixture`, we can use the `Thing` template to define its `vocabWords`, name and `desc`; but since a `Door` is also a kind of `Passage` (which we'll meet below), we can also assign the very common `masterObject` property using `->` in the template before the `vocabWords` property, like this:

```
+ frontDoorOutside: Door ->frontDoor 'front door*doors' 'front door'
;
```

The doors we defined above will start out closed. If we want them to start out open, we should define `initiallyOpen = true` on the `frontDoor` (the master object).

As mentioned above, a `Door` is of class `Openable`. That means that the player can open and close it using the **open** and **close** commands (unless it's locked, of course, but we'll leave locks until a later chapter). It also means that we can test whether it's open or closed by looking at the value of its `isOpen` property. E.g.:

```
hall: Room 'Hall'
     desc()
     {
          "The front door stands ";
          if(frontDoor.isOpen)
              "open";
          else
              "closed";
         "to the north. ";
     }
    north = frontDoor
;
```

We could in fact produce this effect with much briefer code, but it demonstrates the principle (and also demonstrates that the `desc` property can be a more complex method and not just a double-quoted string).

Note that although we can *test* the value of the `isOpen` property, we should never try to change it directly (with a statement like `isOpen = true;)`, either on a `Door` or on any other `Openable` object (since doing so would be liable to break the mechanism that keeps both sides of the same door in sync). Instead we should use the `makeOpen()` method; `makeOpen(true)` to open something and `makeOpen(nil)` to close it.

Exercise 8: Add some doors to the map you've been building (or make a new map and put some doors in it). Observe what happens when you try to make the player character go through a closed door without explicitly opening it first.

4.2 Coding Excursus 5 – Two Kinds of String

So far we've been using double-quoted and single-quoted strings without explaining what the difference is between them, apart from simply stating that some properties need to use single-quoted strings and other properties need to use double-quoted strings. The time has come to explain the difference.

In a nutshell, it's this: *a single-quoted string is a piece of textual data, while a double-quoted string is an instruction to display a piece of text on the screen.*

The difference can be illustrated by the following fragment of code:

```
myObj: object
   myMethod()
   {
     name = 'elephant water';
     "That smells unpleasant! ";
   }
   name = 'green pea'
;
```

When the `myMethod` method of `myObj` is executed, the `name` property (of `myObj`) is changed to 'elephant water' and the text "That smells unpleasant!" is displayed on the screen.

We can also display the value of a single-quoted string on the screen by using the `say()` function:

```
myObj: object
   myMethod()
   {
     name = 'elephant water';
     say('That smells unpleasant! ');
     say(name);
   }
   name = 'green pea'
;
```

Note, by the way, how we generally leave a spare space at the end of a string that we plan to display; that's to ensure that if another piece of text is displayed immediately after it we have proper spacing between the two sentences.

But to return to the distinction between single and double-quoted strings, the apparent exception to the rule that a single-quoted string is a piece of data, while a double-quoted string is an instruction to display something is that many object properties (such as `desc` and `specialDesc`) are defined as double-quoted strings. But this anomaly is more apparent than real. Perhaps the best way to explain it is to say that a property defined as a double-quoted string is short-hand way of defining a method that displays that string; so for example:

```
    desc = "A humble abode, but mine own. "
```

is exactly equivalent to defining:

```
    desc()  { "A humble abode, but mine own. ";   }
```

or indeed:

```
    desc()  { say('A humble abode, but mine own. ');  }
```

And indeed, wherever any TADS 3 documentation suggests that we need to define an object property as a double-quoted string, it's always perfectly legal to define a method (which can be as complicated as we like) that displays something.

Similarly whenever any TADS 3 documentation suggests that we need to define a property containing a single-quoted string, it's always perfectly legal to define a method (which can be as complicated as we like) that returns a single-quoted string; e.g.:

```
    name()
    {
        if(weight > 4)
            return 'heavy ball';
        else
            return 'light ball';
    }
```

Although we can define the initial value of a property to be a double-quoted string, we can't go on to change that property to be another double-quoted string (or anything else for that matter). The following code is illegal:

```
    changeDesc()
    {
        desc = "It's a great big heavy ball. "; // DON'T DO THIS!
    }
```

On the other hand, it's always perfectly okay to change the value of a single-quoted string property. We can also manipulate single-quoted strings in all sorts of other ways. For example:

```
    name = 'black ' + 'ball';
```

Changes the value of name to 'black ball'. We can also write:

```
     name += ' pudding';
```

To append ' pudding' to the end of whatever was in `name`. Other things we can do with single-quoted strings include:

- `str1.endsWith(str2)` – tests whether `str1` ends with `str2` (e.g. if `str1` is 'abcdef' and `str2` is 'def' this will return true).
- `str1.startsWith(str2)` – tests whether `str1` starts with `str2` (e.g. if `str1` is 'abcdef' and `str2` is 'abc' this will return true).
- `str.length()` returns the number of characters in `str` (e.g. if `str` is 'abcdef' this would return 6).
- `str1.find(str2)` tests whether the string `str2` occurs within the string `str1`, and if so returns the starting position of `str2` within `str1` (e.g. if `str1` is 'antique dealer' then `str1.find('deal')` would return 9 while `str.find('money')` would return nil).
- `str.toUpper()` returns a string with all the characters in `str` converted to upper case (e.g. `'Fred Smith'.toUpper` returns 'FRED SMITH').
- `str.toLower()` returns a string with all the characters in `str` converted to lower case (e.g. `'Fred Smith'.toLower` returns 'fred smith').
- `str.substr(start)` returns a string starting at the *start* character of `str` and running on to the end of the string (e.g. if `str` is 'blotting paper' then `str.substr(5)` would return 'ting paper').
- `str.substr(start, length)` returns a string starting at the *start* character of `str` and continuing for no more than *length* characters (e.g. if `str` is 'blotting paper' then `str.substr(5,4)` would return 'ting').

For a full list of the methods available for manipulating single-quoted strings, see the "String" chapter under in Part IV of the *TADS 3 System Manual*.

It may seem that while we can manipulate a single-quoted string in all sorts of ways, if we want to manipulate the contents of a double-quoted string then we're out of luck. That's almost true – after all a double-quoted string is basically an instruction to display something, not a piece of data – but there is a little trick we can use to convert a double-quoted string to a single-quoted one, or rather to capture the output of a double-quoted string in a single-quoted one. For example, suppose we wanted to do something with the `desc` property of some object; we can use code like this to capture the value of the `desc` property (a double-quoted string) in a single-quoted string:

```
    local str = mainOutputStream.captureOutput({: desc });
```

And though we can't change the value of `desc` in any way, we're then free to do whatever we like with `str` (which now contains the same characters as `desc`, but in a single-quoted string). We haven't yet gone far enough into TADS 3 coding to explain exactly *why* this odd-looking construction works; for now, just treat is as an incantation that produces the desired effect!

4.3 Other Kinds of Physical Connector

It's quite common for a door to lead from one location to another, but doors are not the only kind of physical connection that can do this. Other examples include stairways, paths and passages, and TADS 3 has classes to model all of these.

The base class for all these kinds of connector is `Passage.` We use `Passages` in the same way we use `Doors`; that is we define one end of the `Passage` in one location, and the other end in the other location. We then link the two ends of the `Passage` by setting the `masterObject` property of one (not both) of the two `Passage` objects to the other `Passage` object. Finally, we make sure that the relevant directional properties in the two rooms point to the `Passage` objects. We have already seen an example of that in the way `Doors` are set up.

It's unlikely that we'll want to use the `Passage` class directly in a game; we're more likely to want to use one of its subclasses. These are:

- `Stairway –` A `Stairway` is something we can climb up or down. We typically won't use the Stairway class itself, but rather its two subclasses: `StairwayUp` and `StairwayDown`. In a typical set up we use a `StairwayUp` at the lower end and a `StairwayDown` at the upper end. Although we can (and often will) use these classes for flights of stairs, we can also them for anything that an actor might climb up and down, such as hillsides, trees and masts.
- `ThroughPassage` – This is a passage that an actor can go through with an **enter** or **go through** command. We might typically use this for passages, corridors and tunnels.
- `Door` – We've already discussed `Doors` above; in essence they're a kind of `ThroughPassage` that can also be opened and closed.
- `AutoClosingDoor` – A kind of `Door` that automatically closes after someone has gone through it.
- `SecretDoor` – An object that acts as a `Door` but doesn't look like a door until it's opened, for example a bookcase that can be opened to hide a secret passage behind.
- `HiddenDoor` – An object that acts as a Door but is not even visible until it's opened, for example a seamless panel in a wall.
- `ExitOnlyPassage` – An object that represents the exit from a passage that can only be traversed in one direction, for example the bottom end of a chute.
- `PathPassage` – An object representing one end of a path, street, road or other unenclosed passage that one would think of travelling along rather than

through. We can go along such passages with commands such as **follow path** or **take path**.

All these classes are used to represent physical objects at both ends of the connection, the kinds of object that would typically be listed in a room description but not listed separately, for example "A broad flight of stairs leads up to the east" or "A narrow passage leads off to the south" or "A path runs southwest round the side of the house". They are particularly useful when we want an object to represent these kinds of connection between locations without wanting to implement them as locations in their own right (perhaps nothing interesting happens in the passage, so we don't actually want a passage room). A further example might help to illustrate their use:

```
cellar: Room 'Cellar'
    "This cellar is mainly empty apart from a pile of useless junk in the
     corner. The only way out is back up the stairs. "
    up = cellarStairs
;

+ cellarStairs: StairwayUp ->hallStairs 'stairs' 'stairs'
    isPlural = true
;

+ junk: Decoration 'useless pile/junk' 'pile of useless junk'
   "The accumulated rubbish of decades. "
    notImportantMsg = 'None of it is of any conceivable use. '
;

hall: Room 'Hall'
   "The hall is large and bare. A flight of stairs leads down to the south,
    and a long passage leads off to the west. "
    south = hallStairs
    down asExit(south)
    west = hallPassage
;

+ hallPassage: PathPassage 'long passage' 'long passage'
    "The long passage leads off to the west. "
;

+ hallStairs: StairwayDown 'flight/stairs' 'flight of stairs'
;

kitchen: Room 'Kitchen'
   "This is pretty typical kitchen, if a little old-fashioned. A long passage
    leads off to the east. "
    east = kitchenPassage
;

+ kitchenPassage: ThroughPassage ->hallPassage 'long passage' 'long passage'
;
```

There are a few extra points to note here. First, note that the definition of the `cellarStairs` object includes the line `isPlural = true`. Since 'stairs' is plural this ensures that any library message referring to this object treats them as plural, e.g. "You can't take those" rather than "You can't take that."

Second, note that in the definitions of the `junk` and `hallStairs` objects the `vocabWords` don't include the word 'of' even though the player might well refer to them as 'pile of junk' or 'flight of stairs'. This is because the parser will automatically recognize a phrase like 'X of Y' as referring to an object provided that X and Y are both defined as nouns in that object's `vocabWords` property.

Third, note the use of `asExit()` in the definition of the `hall`. This allows two or more exits (in this case south and down) to behave in the same way with only one of them being listed in the exit-lister. In this case the player might reasonably type either **down** or **south** to go down the stairs, so we want both to work, but we wouldn't want both **down** and **south** to appear in the list of exits, since this might mislead the player into supposing they were two separate exits.

There are four more classes (all subclasses of `TravelConnectorLink`) that at first sight look rather like the kinds of connector we've just been discussing, but are in fact something a little different. These are:

- `Enterable` – an object that exists in one location, and which can be entered to take an actor to another location. `Enterables` are used for things such as the outsides of buildings, so that the building can have a presence on its outside and can be entered via a command like **go into building**.
- `EntryPortal` - this is just like an `Enterable`, except that **go through** also works on it. We might typically use it for things like archways and doorways.
- `Exitable` – this is like an `Enterable`, except that you exit it rather than enter it. This can be used for objects representing the current location as an enclosure (a jail cell), or an exit door.
- `ExitPortal` – this is just like an `Exitable`, except that **go through** also works on it. It might be used for a doorway or an archway leading out of somewhere.

There are four important differences between these four classes and the `Passage`-type connectors we looked at previously:

1. `Passage`-type connectors are always used in pairs, with one in each of the two locations being connected, whereas `TravelConnectorLink` objects can be and generally are used singly.
2. One of the two Passage-type connectors in a pair must always be linked to the other via its `masterObject` property. `TravelConnectorLinks` are never linked in this way even if we happen to define one in both of the connected rooms.
3. Where a Passage leads is determined by its `masterObject` property (which indirectly identifies the location of the other end of the Passage). Where a `TravelConnectorLink` leads to is determined by its `connector` property, which defines the `TravelConnector` (which may simply be a room) that will actually

be used when an actor enters, exits, or goes through the **TravelConnectorLink** object.

4. If we define a **Passage** type object (or, more generally, a **TravelConnector**-type object, which we'll be discussing below), we should always point the appropriate direction property of its room to it. We never do this with **TravelConnectorLink** objects.

The possibility for confusion is further increased for newcomers to TADS 3 by the fact that the **TravelConnectorLink** template looks just like the **Passage** template; they both start with an object preceded with **->**. But whereas the **->** part of the template defines the **masterObject** property of a Passage, it defines the connector property of a **TravelConnectorLink**.

Probably the most commonly used of the **TravelConnectorLink** classes is **Enterable**, so we'll use this class to illustrate all this. Suppose we are defining a front drive location which mentions a large house to the south. Without using any templates at all, our definition might look like this:

```
frontDrive: OutdoorRoom
    roomName = 'Front Drive'
    desc = "The drive is impressive, but not half as impressive as the large
            Georgian house that stands directly to the south. "
    south = frontDoor
;

+ house: Enterable
    connector = frontDoor
    vocabWords = 'large georgian house/building'
    name = 'large Georgian house'
    desc = "It has a white-painted front door. "
;

+ frontDoor: Door
    masterObject = frontDoorInside
    vocabWords = 'front white painted white-painted door*doors'
    desc = "It has been painted white. "
;
```

Using templates, this becomes:

```
frontDrive: OutdoorRoom 'Front Drive'
    "The drive is impressive, but not half as impressive as the large
     Georgian house that stands directly to the south. "
    south = frontDoor
;

+ house: Enterable ->frontDoor 'large georgian house/building'
      'large Georgian house'
      "It has a white-painted front door. "
;

+ frontDoor: Door ->frontDoorInside
    'front white painted white-painted door*doors'
    "It has been painted white. "
```

```
;
```

It may seem that in some cases you could use either Passages or **TravelConnectorLinks** to produce the same effect, and this is in fact the case. From the player's point of view the following two examples would behave identically. Using **TravelConnectorLinks**:

```
hall: Room 'Hall'
   "A long passage leads off to the south. "
    south = kitchen
;

+ hallPassage: EntryPortal ->kitchen 'long passage/corridor' 'long passage'
   "It leads off to the south. "
;

kitchen: Room 'Kitchen'
  "A long passage leads off to the north. "
   north = hall
;

+ kitchenPassage: EntryPortal ->hall 'long passage/corridor' 'long passage'
    "It leads off to the north. "
;
```

And the same example using **Passages**:

```
hall: Room 'Hall'
   "A long passage leads off to the south. "
    south = hallPassage
;

+ hallPassage: ThroughPassage ->kitchenPassage 'long passage' 'long passage'
   "It leads off to the south. "
;

kitchen: Room 'Kitchen'
  "A long passage leads off to the north. "
   north = kitchenPassage
;

+ kitchenPassage: ThroughPassage 'long passage' 'long passage'
    "It leads off to the north. "
;
```

Actually, if you tried compiling and running these two examples you might just find one small difference between them. In the second case if the player typed **go through passage** followed by **examine it** the parser would recognize that **it** should now refer to the other side of the passage, and would display an appropriate description. If you did this with the first example, however, the parser would think that **it** still referred to the **EntryPortal** in the room that player character had just left, and so would display a message saying that **it** no longer referred to anything present. To use the technical vocabulary of the TADS 3 library, the library automatically recognizes that the two ends of a **Passage** are *facets* of the same physical object and will resolve pronouns accordingly, but there's no such recognition in the case of **TravelConnectorLinks**, which are taken to be physically distinct objects. We could fix this if we wanted to by using the **getFacets** property of each **EntryPortal** to point to the other **EntryPortal** like this:

```
hall: Room 'Hall'
   "A long passage leads off to the south. "
    south = kitchen
;

+ hallPassage: EntryPortal ->kitchen 'long passage/corridor' 'long passage'
   "It leads off to the south. "
   getFacets = [kitchenPassage]
;

kitchen: Room 'Kitchen'
  "A long passage leads off to the north. "
   north = hall
;

+ kitchenPassage: EntryPortal ->hall 'long passage/corridor' 'long passage'
    "It leads off to the north. "
    getFacets = [hallPassage]
;
```

But this is more work, and correctly suggests that in this particular kind of situation (representing a passage between two locations) we're really better off using **Passages** rather than **TravelConnectorLinks**. As we'll see below, there are even more reasons for preferring **Passages** in this situation, since they make it easier to do a number of things we can't so readily do with **TravelConnectorLinks**.

Exercise 9: Now that we've covered both **Passages** and **TravelConnectorLinks**, look up both these classes in the *Library Reference Manual*. Take a look at the properties and methods they define, and also the list of their subclasses. Then use the *Library Reference Manual* to explore these subclasses.

Exercise 10: Add some Passage and TravelConnectorLink objects to your practice map (or create a new map for the purpose). Compile your game and try it out to make sure that everything works as you expect.

4.4 Coding Excursus 6 – Specials Things to Put in Strings

It may have occurred to you that there's a problem with putting a single-quote mark (or apostrophe) inside a single-quoted string, since if we write something like:

```
   local var = 'dog's dinner'; // THIS IS WRONG
```

The apostrophe in "dog's dinner" will look like the termination of the string, and the code simply won't compile. We get round this problem by using an *escape* character, that is a character that warns the compiler to treat the character that follows it in a special way. In TADS 3 the escape character is the backslash (\). This lets us include a single-quote (or apostrophe) in a single-quoted string by preceding it with a backslash:

```
    local var = 'dog\'s dinner'; // but this is fine
```

We can similarly use the backslash to include a double-quote mark in a double-quoted string:

```
    "\"Right,\" says Fred, \"That's quite enough of that, I think!\"";
```

Note that in this case there's absolutely no need to escape the apostrophe in "That's" because it occurs inside a *double*-quoted string (although it won't do any harm if we do escape it by preceding it with a backslash).

There are a few other characters that have a special meaning when preceded by a backslash. Here's the complete list:

- `\"` - a double quote mark.
- `\'` - a single quote mark (or apostrophe).
- `\n` – a newline character.
- `\b` – a "blank" line (i.e. paragraph break).
- `\^` - a "capitalize" character; makes the next character capitalized.
- `\v` - a "miniscule" character; makes the next character lower case.
- `\ ` - a quoted space (useful if we want to force a certain number of spaces despite the output formatters well-meaning attempts to tidy them up for us).
- `\t` – a horizontal tab.
- `\uXXXX` – the Unicode character XXXX (in hexadecimal digits)

There are also a number of special characters we can use in both single- and double-quoted strings:

- `<.p>` - single paragraph break
- `<.p0>` - cancel paragraph break
- `<./p0>` - cancel `<.p0>`
- `<q>` - smart typographical opening quote ‘ or “
- `</q>` - smart typographical closing quote ’ or ”

These require a few words of further explanation. At first sight `<.p>` may appear to do the same thing as `\b`, but there is a difference. A run of multiple `\b` characters will produce multiple blank lines, will a run of consecutive `<.p>` tags will produce only a single blank line. This means, for example, we can end one string with `<.p>` and begin another with `<.p>` knowing that we'll only get one blank line between them even if the second is displayed directly after the first. The zero-spacing paragraph (or 'paragraph-swallowing tag') `<.p0>` suppresses any paragraph break that immediately follows. We can use it at the end of a string to force the next string to be displayed directly after it without a paragraph break even if the next string starts with `<.p>`. Finally, we can use `<./p0>` at the start of a string to force a paragraph break even if the immediately preceding string ended with a `<.p0>`.

The smart typographical tags `<q>` and `</q>` work by alternating between double and single quotation marks. So for example, if we included the following in our code:

```
   "<q>Right,</q> Fred declares, <q>That's quite enough <q>clever</q>
    talk for now.</q> ";
```

What we'd see displayed is:

"Right," Fred declares, "That's quite enough 'clever' talk for now."

This is useful, but not entirely problem-free. For one thing, the straight apostrophe (') in "That's" now looks badly out of place. This can be cured by including Stephen Granades's cquotes extension in your project (this extension comes with TADS 3; you should find it under ../lib/extensions under the directory where TADS 3 is installed on your system). A less immediately obvious problem is that in a conversation-heavy game it's almost inevitable that we'll miss out a `<q>` or a `</q>` somewhere, and once that happens the 'smart' quotes will thereafter start doing the opposite of what we want. Of course this can often be cured with sufficiently diligent debugging, or with sufficient ingenuity, by writing your own output filter to keep track of smart quotes, but what some TADS 3 authors prefer to do is to define their own typographical quote tags, for example:

```
dquoStyleTag: StyleTag 'q'
  openText  = '&ldquo;'
  closeText = '&rdquo;'
;

squoStyleTag: StyleTag 's'
  openText  = '&lsquo;'
  closeText = '&rsquo;'
;
```

With those style tags defined (don't worry if you don't fully understand how they work, for now you can just copy them into your code) you can use `<.q>` and `<./q>` knowing that they will always produce typographical (curly) opening and closing double-quotes, and `<.s>` and `<./s>` knowing that they'll always produce opening and closing typographical single quotes. Whether you prefer this to using `<q>` and `</q>` is up to you.

It's also worth mentioning that TADS 3 will convert a pair of dashes (--) in textual output to an n-dash , and three successive dashes to an m-dash.

Another special kind of thing we can put inside strings is HTML markup (or that version of HTML markup that TADS 3 recognize). For a full account, see *Introduction to HTML TADS* (which is part of the standard TADS 3 documentation set). Some commonly used HTML tags are:

- `<b> ... </b>` - display text in **bold**
- `<i> ... </i>` - display text in *italics*
- `<u> ... </u>` display text underlined
- `<FONT COLOR=RED> ... </FONT>` - display text in red.
- `<a> ... </a>` display text as a hyperlink.

What the last of these actually does depends on what we put in the `href` parameter of the opening `<a>` tag. We *can* make it display a web page or any of other things hyperlinks normally do, but the most common use in a TADS 3 game is to make it execute a command. For example, if the following statement were executed.

```
    "You could go <a href='go north'>north</a> from here. ";
```

Then the player would see something like the following on screen:

You could go north from here.

If the player then clicked on the north hyperlink, the command 'go north' would be copied to the command line and executed. This is so useful that TADS 3 defines a special function, aHref(), which helps us set this up. Instead of using the explicit HTML markup as in the previous example, we could obtain the same effect with:

```
    "You could go <<aHref('go north',' north')>> from here. ";
```

Or even with:

```
    "You could go <<aHref('go north',' north', 'Go north')>> from here. ";
```

Which would cause the explanatory text 'Go north' to be displayed in the status bar at the bottom of the interpreter window when the player hovers the mouse over the hyperlink.

This, incidentally, introduces the final kind of special thing we can put inside strings, although in this case, *only* inside double-quoted strings, namely the special `<< >>` syntax. This is known as an *embedded expression* since it allows us to 'embed' (i.e. include) an expression inside a double-quoted string. If the expression evaluates to a single-quoted string, or displays a string, then that string will be displayed at that point. If the expression evaluates to a number, then the number will be shown. It's not actually necessary for the expression to evaluate it to anything at all; it's perfectly legal (and often useful) for the embedded expression to be a function or method that we use at that point for its other effects (changing the game state in some way).

A typical use of the embedded expression syntax is in conjunction with the `?:` conditional operator, for example:

```
hall: Room 'Hall'
    "The front door lies <<frontDoor.isOpen ? 'open' : 'closed'>> to the
    north. "
    north = frontDoor
;
```

But we could equally well embed a call to a method, property or function, e.g.:

```
hall: Room 'Hall'
    "The front door lies <<frontDoor.openDesc>> to the north. "
    north = frontDoor
;
```

Strictly speaking, this doesn't do anything we couldn't do without it, since the previous example could be written as either:

```
hall: Room 'Hall'
    desc()
    {
       "The front door lies;
       say(frontDoor.openDesc);
       "to the north. ";
    }
    north = frontDoor
;
```

Or:

```
hall: Room 'Hall'
    desc()
    {
       say ('The front door lies' +  frontDoor.openDesc + 'to the north. ');
    }
    north = frontDoor
;
```

But using the embedded expression is obviously more convenient. Indeed, it is so very convenient that it's a very frequently used feature of TADS 3.

Note that we can't *directly* embed one `<< >>` expression inside another, though we can do so *indirectly* (that is, it's perfectly legal for a `<< >>` embedded expression to call a function or method that itself displays a double-quoted string using a `<< >>` embedded expression).

4.5 TravelConnectors

We've already seen that a `Door` is a special kind of `Passage`. As it so happens, a `Passage` is a special kind of something else, namely a `TravelConnector`. All the kinds of `TravelConnector` we've so far have been physical objects (doors, paths, stairs, corridors and the like), but it's perfectly possible (and often useful) to employ abstract `TravelConnectors` to control travel from one location to another even when there's no such physical object involved. The common reasons for wanting to do this are:

- carrying out some side-effect of travel, such as displaying a message describing the travel.
- imposing some condition that determines whether or not the travel is to be allowed, e.g. the player might be able to squeeze through the narrow passage by himself, but not when he's carrying the bulky box or pushing the large trolley.

Except when the player character (or any other actor) is transported round the map by authorial fiat (e.g. using `me.moveIntoForTravel(someDestination)`) travel round a TADS 3 game map is *always* via a `TravelConnector`. Whenever the player enters a movement command, whether it be a compass direction like **north** or **southwest**, or a command like **climb stairs** or **go through red door**, the library first determines what the relevant `TravelConnector` is and then translates the player's command into `TravelVia conn` (where `conn` is the `TravelConnector` in question). This action in turn works out what the destination of the `TravelConnector` is and then moves the actor there (the full process is actually a bit more complicated than that, but the simple explanation will do for now).

A seeming exception to the rule that travel is always via a TravelConnector is where a directional property points directly to another room, e.g.:

```
hall: Room 'Hall'
   "The kitchen lies to the south. "
    south = kitchen
;

kitchen: Room 'Kitchen'
   "The hall lies to the north. "
   north = hall
;
```

But the exception is only an apparent one, since ***`Rooms`*** *are also* ***`TravelConnectors`****.* That is, `TravelConnector` is one of the several classes from which `Room` inherits. A `Room` is a `TravelConnector` that always leads to itself.

`TravelConnector` defines a number of methods (and properties). The three most important ones to know about are:

- `canTravelerPass(traveler)` – determines if the traveler is allowed to pass through this `TravelConnector` (return `nil` to disallow travel or `true` to allow it).
- `explainTravelBarrier(traveler)` – if `canTravelPass()` prevents travel, this method is used to display a message explaining why the traveller can't pass.

- **noteTraversal(traveler)** – carry out any side effects of the travel, e.g. by displaying a message describing it.

Some additional methods and properties it's also quite useful to know about are:

- **connectorStagingLocation** – the place an actor needs to be before travelling via this **TravelConnector** (this won't make much sense till we come to look at **NestedRooms**).
- **isCircularPassage** – if this is true then the travel will be described even if it leads back to its starting point; otherwise such circular travel won't be treated as real travel.
- **isConnectorListed** – determines whether this **TravelConnector** is to be listed in any list of exits.
- **travelBarrier** – a single **TravelBarrier** object, or list of **TravelBarrier** objects, that applies to this **TravelConnector** (we'll say more about **TravelBarriers** below).
- **getDestination(traveler, origin)** – determines where this **TravelConnector** leads to. Most subclasses of **TravelConnector** define this in a way that effectively looks after itself.

There are other methods and properties besides; if you want the full story look up **TravelConnector** in the *Library Reference Manual*.

Since **Passage** is a kind of **TravelConnector**, we can illustrate some of these methods on the kind of **Passage** objects we've already seen:

```
cave: Room 'Cave'
   "A narrow tunnel leads south. "
   south = narrowTunnel
;

+ narrowTunnel: ThroughPassage 'narrow passage*passages' 'narrow passage'
   "It looks only just wide enough for you to squeeze through. "

    canTravelerPass(traveler)
    {
        return !bigHeavyBox.isIn(traveler);
    }

    explainTravelBarrier(traveler)
    {
       "You'll never get through the narrow passage carrying that big heavy
       box! ";
    }

    noteTraversal(traveler)
    {
       "You just manage to squeeze through the narrow tunnel. ";
    }
;
```

We're unlikely to use the `TravelConnector` base class itself, but we should now become acquainted with its other subclasses, each representing some kind of non-physical connection between rooms:

- `RoomConnector` – connects two `Rooms` together (the rooms being specified in the `room1` and `room2` properties of the `RoomConnector`). We seldom need to use one of these, since we'd normally just connect the two rooms together directly, but we might want to use a `RoomConnector` if we wanted to enforce some kind of travel barrier between the two rooms.
- `OneWayRoomConnector` – as its name suggests, this is a `RoomConnector` that works in one direction only; we specify where it leads to by using its `destination` property (which can be defined with the -> symbol in its template).
- `RoomAutoConnector` – we're unlikely ever to need to define one of these in our own code, but it's in fact the type of `TravelConnector` from which Room inherits, and which allows a Room to be the target of a directional property.
- `TravelMessage` – this is in many ways similar to a `OneWayRoomConnector`, except that it displays a message when it's traversed. As with `OneWayRoomConnector` we use the destination property to define where the `TravelConnector` goes. We use `travelDesc` to display a message for when the player character travels via this connector, and `npcTravelDesc` for the message (if we want one) that's displayed when an NPC does so.
- `DeadEndConnector` – this is a special kind of `TravelMessage` that takes us back to where we started. We use this to represent the kind of situation where the player character sets off in a particular direction but then turns back, either because s/he encounters some kind of obstacle, or because s/he fails to find anything interesting.
- `NoTravelMessage` – this is a special kind of `TravelMessage` which disallows travel and explains why it is not possible (used when there's some kind of *physical* reason preventing travel, e.g. there's a solid wall in the way, or walking that way would take the player off the edge of a cliff).
- `FakeConnector` – this is a special kind of `NoTravelMessage` which disallows travel and explains why the player character is unwilling to attempt it (even though it is apparently possible). We'd use this when there's a *motivational* reason preventing travel (e.g., you don't want to leave town until you've solved the mystery of the missing psychic piglet).
- `AskConnector` – a kind of connector to use when there's more than one exit in a certain direction, e.g. two doors in the north wall. Precisely how this works is beyond the scope of this book; for details see the *Library Reference Manual*.
- `TravelWithMessage` - A mix-in class that can be added to objects that also

inherit from **TravelConnector** to add a message as the connector is traversed. Note that this isn't itself a travel connector; it's just a class that should be combined with **TravelConnector** or one of its subclasses. This class should be in the superclass list before the **TravelConnector**-derived superclass. Like **TravelMessage** (which inherits from it) this class defines **travelDesc** and **npcTravelDesc** properties to display the traversal messages.

A few further words of explanation may be in order. With the exception of **NoTravelMessage**, (and **SecretDoor** and **HiddenDoor** when closed) every kind of **TravelConnector** shows up in the exit lister (unless we override this behaviour). This is the only real difference between **NoTravelMessage** and **FakeConnector**. We use a **NoTravelMessage** to remind a player or explain why travel in a certain direction is (probably quite obviously) physically impossible; we use a **FakeConnector** to represent a direction in which travel looks perfectly possible but is going to be disallowed for motivational reasons (usually to provide a 'soft boundary' to the map).

The difference between **FakeConnector** and **DeadEndConnector** is perhaps more subtle; we use the first when travel is not even going to be attempted ('You don't want to leave town until you've found the missing psychic piglet') and the second when travel is attempted but the player character turns round and comes back ('You walk a few hundred yards down the road, but seeing nothing but desert for miles and miles ahead, you turn round and come back' or 'You walk a few dozen metres down the forest track until you find a fallen tree completely blocking your path, forcing you to retrace your steps'). The reason we need to make this distinction is that other things, and in particular other actors, may respond to the player character's attempts at travel; they should react in the second case, but not the first.

It might seem that using these various types of connector might be fairly cumbersome, as if (on analogy with the various **Passage** objects), one would have to do this kind of thing:

```
forestClearing: OutdoorRoom 'Forest Clearing'
    "A variety of paths runs of in all sorts of directions. "
    north = forestOneWay
    east = forestFake
    west = forestDeadEnd
    south = forestTravelMsg
    southeast = forestNoTravel
;
```

forestOneWay: OneWayRoomConnector

```
  destination = byStream
   canTravelerPass(traveler)   {   return boots.isWornBy(traveler);  }

   explainTravelBarrier(traveler)
   {
       "That way is too muddy to walk down without a pair of sturdy boots. ";
   }
;
```

```
forestFake: FakeConnector
   travelDesc = "That way looks so dark and threatening you really don't
    fancy it. "
;

forestDeadEnd: DeadEndConnector
   travelDesc = "You set off down the track, but shortly encounter a fallen
    tree that completely blocks your path, forcing you to turn round and come
    back. "
;

forestTravelMsg: TravelMessage
   travelDesc = "You walk briskly along the path for a couple of hundred
    yards before coming to another junction. "
   destination = forestJunction
;

forestNoTravel: NoTravelMessage
  travelDesc = "The trees are too densely packed in that direction. "
;
```

It would indeed be tedious and verbose to have to do this (although it would work), but fortunately we don’t have to. The code can be made much more concise by using a combination of the appropriate templates and *anonymous nested objects*. We’ll explain anonymous objects in more detail in the next chapter, but for now all we need to know is that we don’t need to give every object a name (so it can be *anonymous*) and that we can define an anonymous object directly as the value of the property of another object, in which it is said to be *nested*. Using these two techniques together we can compress the previous example to:

```
forestClearing: OutdoorRoom 'Forest Clearing'
    "A variety of paths runs of in all sorts of directions. "
    north: OneWayRoomConnector
    {
      -> byStream
      canTravelerPass(traveler) { return boots.isWornBy(traveler); }
      explainTravelBarrier(traveler)
      {
       "That way is too muddy to walk down without a pair of sturdy boots. ";
      }
    }

    east: FakeConnector { "That way looks so dark and
       threatening you really don't fancy it. " }
    west: DeadEndConnector {"You set off down the track, but shortly
       encounter a fallen tree that completely blocks your path, forcing
       you to turn round and come back. " }
    south: TravelMessage { ->forestJunction
      "You walk briskly along the path for a couple of hundred
      yards before coming to another junction. "
    }
    southeast: NoTravelMessage { "The trees are too densely packed in that
           direction. "}
;
```

Contrary to possible appearance, we haven't actually reduced the numbers of objects involved by doing this, we've just defined them much more succinctly. Also, by keeping everything together on the **forestClearing** object, we've probably made it much easier to see what's going on.

In this example, the player character is preventing from going north from the clearing unless s/he's wearing the boots. If we had several muddy paths on which we wanted to impose the same condition, it would be tedious to have to code essentially the same thing on all the relevant **TravelConnectors**. An alternative is to define a **TravelBarrier** object, e.g.:

```
bootBarrier: TravelBarrier
      canTravelerPass(traveler) { return boots.isWornBy(traveler); }
      explainTravelBarrier(traveler)
      {
       "That way is too muddy to walk down without a pair of sturdy boots. ";
      }
;
```

Then we can just attach this **bootBarrier** object to every **TravelConnector** to which it applies, e.g.:

```
forestClearing: OutdoorRoom 'Forest Clearing'
    "A variety of paths runs of in all sorts of directions. "
    north: OneWayRoomConnector
    {
      -> byStream
      travelBarrier = bootBarrier
    }
;
```

Another reason to define **TravelBarriers** might be if there were several different reasons for which we might want to block travel on the same connector. Suppose, for example, that we want to stop the player going north either if s/he's not wearing the boots or if s/he's left the map behind. Since we want the message explaining why travel isn't allowed to reflect the reason we're stopping it, it would be convenient to implement them as two different **TravelBarriers** (rather than putting a compound condition in **canTravelerPass()** and then have **explainTravelBarrier()** work out which condition failed before displaying its message):

```
mapBarrier: TravelBarrier
      canTravelerPass(traveler) { return map.isIn(traveler); }
      explainTravelBarrier(traveler)
      {
       "You'd better not go any further that way without a map. ";
      }
;

forestClearing: OutdoorRoom 'Forest Clearing'
    "A variety of paths runs of in all sorts of directions. "
    north: OneWayRoomConnector
    {
      -> byStream
      travelBarrier = [bootBarrier, mapBarrier]
    }
;
```

This uses a feature of the TADS 3 language we haven't been formally introduced to yet, namely a list. All we need to know about lists at the moment is that they are special data type that allows us to group a number of items together, that they're enclosed in square brackets, and that list elements must be separated by commas. We'll fill in more details later.

There are a couple of specials kinds of `TravelBarrier`:

- `PushTravelBarrier`: a `TravelConnector` that allows regular travel, but not travel that involves pushing something. By default, we block all push travel, but subclasses can customize this so that we block only specific objects.
- `VehicleBarrier`: a `TravelConnector` that allows actors to travel, but blocks vehicles. By default, we block all vehicles, but subclasses can customize this so that we block only specific vehicles.

Exercise 11: Let's take it for granted now that you'll look up these `TravelConnectors` and `TravelBarriers` in the *Library Reference Manual*, and carry straight on with suggesting a game you can implement to try them out.

Try creating a game based on the following specification. The game starts in a hall, from which there are four exits. One exit leads down via a flight of stairs to a cellar. One leads south via a path to the kitchen. One leads north through the front door. And one leads east directly into the lounge, but the description of the hall suggests that you go through an archway to get there.

From the kitchen a passageway leads north back to the hall, but there's a secret panel to the east and a laundry chute to the west (you can go down the chute but not back up it). In the kitchen is a trolley that can be pushed around, but it can't be pushed up and down stairs, or indeed through the laundry chute. There's also a flashlight in the kitchen which can be used to explore dark rooms.

The cellar is a dark room, from which a flight of stairs leads back up into the hall. On the west side of the cellar is the bottom end of the laundry chute, from which the player can only emerge (but no go back up again).

In addition to the exit west back out to the hall (which should describe the player character returning to the hall when s/he goes that way), the lounge has two doors in the south wall, an oak door (which automatically closes after someone goes through it) and a pine door (you'll need to study what the *Library Reference Manual* has to say on `AskConnector` to set this up properly). You can't push the trolley through the pine door.

On the other side of the oak door is a study. On the west wall of the study is a bookcase which is in fact the other side of the secret panel on the east wall of the kitchen (so that opening the bookcase allows direct access between the kitchen and the study).

On the other side of the pine door is a room without a floor, situated above the cellar (so that anything dropped here will fall into the cellar below). Hovering in this chamber are a pair of anti-gravity shoes. Until the player character is wearing the shoes s/he'll need to cling to the door to stop falling down into the cellar.

The front door leads north into a drive. To the north of the drive is a road, but the player character doesn't want to go there. To the west lies a wood that's so dense that if the player character tries to enter it s/he soon has to turn back. An oak tree stands in the middle of the drive and the player can climb it (but obviously can't push the trolley up it). Anything dropped at the top of the tree will fall to the ground below. Meanwhile an old bicycle is leaning against the front of the house; the bike can be ridden but can't go anywhere you can't push the trolley (to implement the bike declare it to be of class **`Chair`**, **`Vehicle`**; we haven't met these classes yet, but you can just use them here).

From the drive a path leads east onto a lawn. To east and south the lawn is enclosed by a bend in the river, but a path leads west back to the drive. Also, there's a boat moored up on the river, and you can board the boat to the east. To the north lies impenetrable shrubbery. If you're wearing the anti-gravity shoes (but not otherwise) you can walk south across the river to a meadow (and back north again across the river to the lawn), but you can't push the trolley or ride the bike across the river.

Boarding the boat from the lawn takes you to its main deck. From there you can go starboard back to the lawn or aft to the main cabin. Once in the main cabin you can go out or forward to the main deck, or port into the sleeping cabin.

Even implementing a game as basic as this may require some features of TADS 3 we haven't encountered yet to do properly, so don't worry if there are some things you can't quite get to work fully. Just see how far you can get. When you've got as far as you feel you can (or want to), you can compare your version with the game connectors.t from the set of sample games.

5 Containment

5.1 Containers and the Containment Hierarchy

5.1.1 The Containment Hierarchy

As we've already seen, we can locate objects (and the player) in rooms by setting their location property, initially in one of three (functionally equivalent ways):

```
redBall: Thing 'red ball*balls' 'red ball'
   location = hall
;

redBall: Thing 'red ball*balls' 'red ball' @hall
;

hall: Room 'Hall'
;

+ redBall: Thing 'red ball*balls' 'red ball'
;
```

Things can also be picked up and moved to other rooms, or moved by author fiat using the `moveInto(newLoc)` method. But there's more to containment than this; both in the real world and in Interactive Fiction objects in be in, on, under or behind other objects, not just in rooms. For now we'll concentrate on objects being inside other objects; we'll expand this to on, under and behind in the next section.

Concretely, objects can be inside certain kinds of other object such as boxes, packing cases, cabinets, drawers, sacks, bags, suitcases and any other kind of object capable of containing other objects. In TADS 3 an object that can contain other objects must be of class `Container`. `Containers` can be nested, that is one `Container` can contain another `Container` which can itself contain other things (including other `Containers`).

Slightly more abstractly, every physical object (i.e. a `Thing` or something derived from `Thing`) in a TADS 3 game has a location property (at least as a first approximation; `MultiLocs` can be in several places at once, but we'll meet them in a later chapter) which defines where it is. This location property will hold either another object or `nil`. If it's nil then either the object is a top-level room, or the object is off the map (we can, for example, use `moveInto(nil)` to move an object out of play). If it's another object then that second object will be a room, an actor (if the actor is carrying or wearing the object) or a `Container` (or one of the other classes we'll meet in the next section).

5.1.2 Moving Objects Around the Hierarchy

During game-play, one object can be placed inside another using the PUT IN command, e.g. put red ball in blue box. We can also move objects in and out containers in the same way as we move then in and out of rooms, e.g. redBall.moveInto(blueBox). We can also use the same technique to move objects in and out of the player's (or another actor's) inventory. For example, redBall.moveInto(me); would cause the player character to end up holding the red ball (assuming the player character has been defined as me).

In principle one might expect to be able to move actors around in the same way; e.g. to teleport one the player from the hall to the cellar you might try:

```
   me.moveInto(cellar); // DON'T DO THIS
```

But this won't work. To move actors around we should use **moveIntoForTravel()** instead:

```
   me.moveIntoForTravel(cellar); // but this is fine
```

For a full account of the do's and don'ts of moving actors around, see the article on 'NPC Travel' in the *TADS 3 Technical Manual* (despite the name of the article, just about everything it says applies equally to player character travel). It's also worth being aware that it's sometime necessary to use **moveIntoForTravel()** to move other objects around as well; for example if you want an object to be magically transported to the inside of a closed container. The general rule of thumb is that if you find that **moveInto()** isn't doing what you want (usually manifested with unwelcome run-time errors) try using **moveIntoForTravel()** instead.

5.1.3 Defining the Initial Location of Objects

We can use one of three methods to define the initial location of objects when some objects are inside **Containers**. Suppose that a small red pen is in a small yellow box which is inside a large blue box which is in the hall. We can first of all set this up by explicitly defining the location property of each of the objects:

```
hall: Room 'Hall'
;

blueBox: Container 'large blue box*boxes' 'large blue box'
   location = hall
;

yellowBox: Container 'small yellow box*boxes' 'small yellow box'
   location = blue box
;

redPen: Thing 'small red pen*pens' 'small red pen'
   location = yellowBox
;
```

Or we can do the same thing more compactly using the @ notation in the template:

```
hall: Room 'Hall';

blueBox: Container 'large blue box*boxes' 'large blue box' @hall;

yellowBox: Container 'small yellow box*boxes' 'small yellow box' @blueBox;

redPen: Thing 'small red pen*pens' 'small red pen' @yellowBox;
```

Or we can do it, slightly more compactly, using an extension of the + notation:

```
hall: Room 'Hall';

+ blueBox: Container 'large blue box*boxes' 'large blue box';

++ yellowBox: Container 'small yellow box*boxes' 'small yellow box';

+++ redPen: Thing 'small red pen*pens' 'small red pen';
```

This last notation is particularly convenient, and also gives quite a good visual representation of what's inside what; it is therefore the containment notation that will be most commonly used in this manual. Another minor advantage of this notation is that if you decide to change the name of an object, you don't need to change the reference to it on all the objects it contains.

Since we'll be seeing a lot of this notation from now on, it's worth explaining it a bit more fully. In general, an object preceded by *n* plus signs is contained within the nearest object above it in the same source file preceded by *n-1* plus signs. The example above is fairly straightforward, since each object has one plus sign than the one before it, so that each object contains the next. A slightly more complicated example might be this:

```
hall: Room 'Hall';

+ blueBox: Container 'large blue box*boxes' 'large blue box';

++ yellowBox: Container 'small yellow box*boxes' 'small yellow box';

+++ redPen: Thing 'small red pen*pens' 'small red pen';

++ greenBox: Container 'small green box*boxes' 'green box';

+++ blackPencil: Thing 'black pencil*pencils' 'black pencil';

+++ whiteFeather: Thing 'white feather*feathers' 'white feather';

++ orangeBall: Thing 'orange ball*balls' 'orange ball';

+ oldHat: Wearable 'old hat*hats*garments' 'old hat';
```

In this example, the blue box and the old hat are both directly in the hall. The yellow box, the green box and the orange ball are all directly in the blue box. The red pen is directly in the yellow box, and the black pencil and white feather are directly in the green box.

While the + notation is very useful for setting up the containment hierarchy, it needs to be used with some care. For example, if we move an object from one location to another (using cut and paste in our source code), we need to make very sure that any + signs still mean what we intend them to mean. Also, it can become tricky to ensure that a containment hierarchy defined with + signs is actually the one we want once it includes long and complex objects, or once we start adding other objects in between the existing ones in our source. While it's generally safe to use the + notation for doors, passages, decorations and simple fixtures in a location, some authors may prefer to define long and complex objects elsewhere in their source code using the @ notation. When it comes to defining all but the very simplest NPCs (non-player characters) this becomes almost essential.

5.1.4 Testing for Containment

We often want to test for containment, and there are four methods (defined on `Thing`, and hence available for all `Things` and anything derived from `Thing`) that help us to do this:

- `isIn(obj)` – determines whether the object this is called on is in *obj*, either directly or indirectly (so in the above example `isIn(hall)` would be true for every object except the hall and `isIn(blueBox)` would be true for the yellow box, the red pen, the green box, the black pencil, the white feather and orange ball.
- `isDirectlyIn(obj)` – determines whether the object this is called on is *directly* in *obj*. In the above example `isDirectlyIn(hall)` would be true for the blue box and the old hat, while `isDirectlyIn(blueBox)` would be true for the yellow box and the red pen.
- `isOrIsIn(obj)` – determines whether the object is either *obj* itself or is directly or indirectly in *obj*. In the above example, `isOrIsIn(hall)` would be true for everything; `isOrIsIn(yellowBox)` would be true for the yellow box and the red pen.
- `isHeldBy(actor)` – determines whether the object is being held by *actor*. For most things this is the same as testing whether the object `isDirectlyIn(actor)`; the difference is that something currently worn by the actor is treated as not held by the actor. So, for example, if the actor is wearing the old hat, `oldHat.isHeldBy(actor)` is `nil` (though `oldHat.isWornBy(actor)` would then be true), but if the actor takes the hat off and continues to carry it `oldHat.isHeldBy(actor)` becomes true. If the actor were to pick up the red pen directly (taking it out of the box) then

`redPen.isHeldBy(actor)` would be `true`, but if the actor took either the blue box or the yellow box, leaving the red pen in the yellow box, then `redPen.isHeldBy(actor)` would be `nil`.

We'd typically use these methods in conditional statements, such as:

```
      if(!orangeBall.isHeldBy(me))
            "You need to be holding the orange ball before you can throw it
            anywhere. ";

      if(whiteFeather.isIn(hall))
            "The white feather must be around here somewhere. ";
```

Just as we can test whether one object is inside another, we can also test what other objects an object directly or indirectly contains. For this we use two properties/methods:

- `contents` – a list of objects *directly* contained by this object.
- `allContents` – a list (technically a `Vector`) of objects directly or indirectly contained by this object.

So, in the above example, `blueBox.contents` would be a list consisting of `yellowBox` and `orangeBall`, whereas `blueBox.allContents` would be a Vector containing everything except the hall, the old hat and the blue box (at this point the difference between a list and a Vector need not detain us; we'll deal with it later on).

Conversely, we may want to know which room an object is in, even though it may be in a container inside a container. For this we use the `getOutermostRoom` method. Everything in the hall would have return hall as the value of `getOutermostRoom`.

5.1.5 Containment and Class Definitions

There's just one more thing to note about the plus notation before we move on to a slightly different topic, and that is how it interacts with class definitions. The short answer is that class definitions are ignored for purposes of the object containment hierarchy, so if we were to write:

```
hall: Room 'Hall';

+ blueBox: Container 'large blue box*boxes' 'large blue box';

++ yellowBox: Container 'small yellow box*boxes' 'small yellow box';

class Pen: Thing '*pens'
   bulk = 2
;

+++ redPen: Pen 'small red pen' 'small red pen';
```

The red pen would still end up inside the yellow box, and the Pen class would be nowhere (it can't be anywhere, since it's not a physical object; it's more akin to the

Platonic idea of a Pen, or an abstract specification of what we want all Pens to have in common).

5.1.6 Bulk, Weight and Container Capacity

In this example we defined `bulk = 2` on the Pen class, and this conveniently leads us into the next point to make about containment. It's unrealistic to allow a large chair to fit inside a small purse, and there may be a limit to the total weight an actor can carry. To model this TADS 3 defines a `bulk` property and a `weight` property on every `Thing`, and a `bulkCapacity` on every container (or actor) and a `weightCapacity` property on every actor. A player cannot insert an object inside a container if doing so would make the total bulk of all objects in that container exceed the `bulkCapacity` of that object. Likewise, an actor cannot pick up an object if doing so would mean that either the total `bulkCapacity` or the total `weightCapacity` of the actor would be exceeded. Since inventory limits are often considered something of an unnecessary nuisance by many players of IF, the library defaults make it very unlikely that either the `bulkCapacity` or the `weightCapacity` of an actor would be exceeded, since these are both set to 10,000, but we can, of course, set them to something rather smaller if we wish. The default `bulkCapacity` of a `Container` is likewise set to 10,000 so unless we make it smaller (either globally or on particular containers) we're unlikely ever to exceed it.

By default the library defines the weight and bulk of every (portable) `Thing` as 1. NonPortable changes these values to 0, on the principle that since NonPortable items aren't generally moved around, their bulk and weight are mostly irrelevant; they usually only become relevant when a `NonPortable` item is part of something else (as a `Component`, say, or a `Decoration` perhaps), in which case we generally don't want them to contribute to the bulk or weight of the item of which they're a part. The one class that is given a bulk of larger than 1 by the standard library is `Person`, which is given a bulk of 10. In a game that wants to model bulk with any precision this value may be rather too small; a person is generally a lot more than ten times the smallest object one might want to carry around (a small pebble, say), so it may be convenient to change this to something bigger (40, 50 or 100 say). This is fine, but if we do this we need to remember to change the `bulkCapacity` of certain `NestedRoom` classes (objects that can contain actors), which are generally set just large enough to contain one or two people, notably `Chair` (we'll talk more about `NestedRooms` in a later chapter).

There is one more property that can limit the bulk of things a player can pick up or put into containers, namely `maxSingleBulk`. Regardless of whether the container would become full, or the player still has room in his notional hands, an object can't be inserted into a container (or picked up by a player) if it's bulk exceeds the `maxSingleBulk` for that container or actor. The default `maxSingleBulk` is relatively small (it's 10), so it's quite easy to exceed it; this is something we often need to watch out for if we're wondering why an object won't fit into a container that seems to be quite big enough for it (or can't be picked up by an actor who still has plenty of bulk capacity left).

There is, in fact, also a Container method that has an effect potentially similar to that of **maxSingleBulk**, namely **canFitObjThruOpening(obj)**. By default this returns true, but it could be used to prevent a large object fitting through a narrow opening (e.g. if we're modelling a large bottle with a narrow neck); for example:

```
bottle: Container 'large narrow bottle/neck*bottles' 'bottle'
      "It's quite large, but its neck is narrow. "
      canFitObjThruOpening(obj)
      {
            return obj.bulk < 3;
      }
;
```

Admittedly this example doesn't do anything much different from setting **maxSingleBulk = 2** on the bottle, but a more sophisticated version might do so (for example if we wanted to take the shape of the object into account as well as the bulk, so that, for instance, we could fit a long narrow object like a pencil through the opening but not an even moderately-sized round one such as ball-bearing, even though the pencil and the ball-bearing might have the same bulk).

There's one further point to note about bulk and weight. The **weight** property gives the weight of the object in itself, but if the object contains other objects then its carrying weight will be its own weight plus the weight of everything it contains (the weight of a sackfull of coal is the weight of the empty sack plus the weight of the coal). To get at this full carrying weight, use the **getWeight()** method. There is also a **getBulk()** method to get at the total bulk of containers; although the bulk of most containers remains constant regardless of what's put inside, that's not true of all containers (a sack that's full of coal is bulkier than an empty sack, and the **StretchyContainer** class we'll be meeting below allows us to model this), so to be sure that our code doesn't contain subtle bugs, it's safer to use **getBulk()** as well as **getWeight()** in calculations involving the bulk and weight of objects.

5.1.7 Items Hidden in Containers

Most of the properties and methods we've just been discussing are defined on the **BulkLimiter** class (which you might want to look up in the *Library Reference Manual*). This is the ancestor both of the **Container** class, and of a number of other classes we'll be looking at below. A further property defined on **BulkLimiter** is **revealHiddenItems**. By default this is true; this means that if we put any **Hidden** items in a **Container** (or other **BulkLimiter**) they're automatically revealed when the player looks in the **Container**.

For example, suppose we decided that the player shouldn't notice the red pen till s/he explicitly looked in the yellow box; we could handle that by defining:

```
++ yellowBox: Container 'small yellow box*boxes' 'small yellow box';

+++ redPen: Hidden 'small red pen*pens' 'small red pen';
```

Provided we didn't override `revealHiddenItems` on the yellow box, the red pen would remain hidden until the player looked in the yellow box, whereupon the red pen would be revealed and listed as part of the yellow box's contents.

5.1.8 Notifications

There are two further methods of `BulkLimiter` (actually they're initially defined on Thing) it's useful to be aware of at this stage, namely `notifyInsert(obj, newCont)` and `notifyRemove(obj)`. These are both called when we use `moveInto()` to move an object to a new location; if we need to we can bypass them by using `baseMoveInto()` instead.

Of these, `notifyRemove()` is slightly the simpler, so we'll deal with it first. Whenever an object is about to be removed from inside another object (whether the first object is directly or indirectly contained in the second), `notifyRemove(obj)` is called on the containing object with the object about to be removed from it as the *obj* parameter. By default this does nothing, but a trivial example will help make it clear how it can be used:

```
blackBox: Container 'black box' 'black box' @hall
    notifyRemove(obj)
    {
        "Removing <<obj.theName>> from <<obj.location.theName>>! ";
    }
;

+ greenBox: Container 'green box*boxes' 'green box'
;

++ pebble: Thing 'pebble/stone*pebbles' 'pebble'
;
```

If the player were to issue the command **take pebble** the game would respond with "Removing the pebble from the green box!". If the player were to issue the command **take green box** the game would respond with "Removing the green box from the black box." Note then, that this method is called just before the movement is carried out. This means that we could, if we wished, use `notifyRemove()` to stop an object being removed from a container:

```
blackBox: Container 'black box*boxes' 'black box' @hall
    notifyRemove(obj)
    {
        if(obj == pebble)
        {
            "The pebble refused to leave <<obj.location.theName>>! ";
            exit;
        }
        else
            "Removing <<obj.theName>> from <<obj.location.theName>>! ";
    }
;

+ greenBox: Container 'green box*boxes' 'green box'
;

++ pebble: Thing 'pebble/stone*pebbles' 'pebble'
;
```

This could result in the following transcript:

>take pebble
The pebble refused to leave the green box!

>take green box
Removing the green box from the black box!

>take pebble
Taken.

One new feature we've just introduced here is `exit`. Technically speaking this is a *macro*, but we haven't met macros yet, so for now we can just think of it as a special statement that stops an action in its tracks.

As suggested above, `notifyInsert()` is a little more complicated. For one thing it's called with two parameters: `notifyInsert(obj, newCont)` where *obj* is the object being moved and *newCont* is the container *obj* is about to be moved into (which may be the object `notifyInsert()` is being called on, or another object contained within that object). This can be illustrated via an extension to our previous example:

```
blackBox: Container 'black box*boxes' 'black box' @hall
    notifyInsert(obj, newCont)
    {
        "Putting <<obj.theName>> in <<newCont.theName>>. ";
    }

    notifyRemove(obj)
    {
        "Removing <<obj.theName>> from <<obj.location.theName>>. ";
    }
;

+ greenBox: Container 'green box*boxes' 'green box'
;

++ pebble: Thing 'pebble/stone*pebbles' 'pebble'
;
```

Which could give us a transcript like:

```
You see a black box (which contains a green box (which contains a pebble)) here.

>take pebble
Removing the pebble from the green box.

>take green box
Removing the green box from the black box.

>put green box in black box
Putting the green box in the black box.

>put pebble in green box
Putting the pebble in the green box.
```

This notification occurs just before the object being moved is inserted into its new container, so once again it could be used to prevent the insertion from going ahead. And this is in fact just what `BulkLimiter` uses to prevent an object that's too big from being inserted when it would result in exceeding the `BulkLimiter's bulkCapacity` or `maxSingleBulk`. But therein lies a potential complication, or rather a potential trap. If we override `notifyInsert()` on a container with our own code, as in the above example, we are in effect replacing what the library does with `notifyInsert()`, in this case thereby removing the check that prevents something too big from being inserted into the container. *This is a very easy mistake to make*.

There is a solution, but it involves a programming construct we haven't yet met. However, since the problem it solves is so very common, it's worth anticipating the full discussion and giving the basic solution now:

Whenever you override a library method, be very sure to call the inherited method unless you are ***very*** *sure that you don't need to*.

So how do we call the inherited method, and what exactly does it mean? We'll give a fuller answer shortly, but for now the short answer is that the inherited method is what the method would have done if we hadn't overridden it, and we call it by using the `inherited` keyword along with the parameter list of the method we're overriding. So, for example, the right way to override `notifyInsert()` in our previous example would have been this:

```
blackBox: Container 'black box*boxes' 'black box' @hall
    notifyInsert(obj, newCont)
    {
        inherited(obj, newCont);
        "Putting <<obj.theName>> in <<newCont.theName>>. ";
    }
;
```

Note that we follow the `inherited` keyword with the parameter list just as it appears in the method definition; indeed it's a good idea to use copy and paste to do this. *It's also a good idea to get into the habit of remembering to call inherited methods from very early on.* We'll expand on this point next.

5.2 Coding Excursus 7 – Overriding and Inheritance

We briefly introduced the concept of inheritance in Coding Excursus 2. The time has come to delve into it a little deeper.

As we have seen, an object can inherit from one or more classes. If we define a new class that too can inherit from one or more classes. In the TADS 3 inheritance model, it's even possible for an object to inherit from another object, or even a for a class to inherit from an object. The following are all perfectly legal definitions:

```
myObj: PresentLater, Thing
;

mySecondObj: myObj
;

class myClass: Container, Fixture
;

class mySecondClass: myObj
;
```

There is, indeed, very little difference between objects and classes in TADS 3, except that:

- classes are not included in the object containment hierarchy (as we have just seen).
- if we write code to iterate over objects, classes will not be included (as we'll see some way below).
- classes are declared using the keyword `class` (as shown in the above example).

Nonetheless, it is still worth observing the distinction between classes and objects; we use classes to define behaviour we want to apply to several relevantly similar objects, and objects to represent concrete instantiations of those classes.

The power of this model lies in the fact that we can not only just inherit the behaviour of classes (or objects), we can also modify and override that behaviour on particular objects and subclasses. If you have not yet done so, now might be a good time to read the article 'Object-Oriented Programming Overview' in the *TADS 3 Technical Manual,* which explains this all in a bit more detail.

The basic procedure for overriding a property or method is straightforward; we simply redefine the property or method on the inheriting object. We effectively do this every time we define a standard property on an object; for example, when we define the **desc** property of a **Thing** we're overriding the library default that would otherwise say "You see nothing unusual about the whatsit." More generally, suppose we define (or use) **MyClass** and then derive **myObj** from it, overriding its **name** and **bulk** properties and its **makeBigger()** method:

```
class Blob: Thing
      bulk = 2
      weight = 2
      makeBigger(inc)   { bulk += inc; }
      makeLighter()
      {
            if(weight > 0)
                  weight-- ;
      }
      name = 'blob'
;

greenBlob: Blob
      bulk = 3
      name = 'green blob'
      makeBigger(inc)
      {
         "\^<<theName>> just got bigger! ";
      }
;
```

With this definition, **greenBlob.bulk** is 3, **greenBlob.weight** is 2, and **greenBlob.name** is 'green blob'. After executing **greenBlob.makeLighter()** once, **greenBlob.weight** will be 1. When we call **greenBlob.makeBigger(2)**, however, the bulk of **greenBlob** won't change, even though we'll see the message "The green block just got bigger!".

This probably wasn't what we wanted; we probably wanted the message to display *and* the bulk of **greenBlob** to grow by 2. We could, of course, just repeat the statement **bulk += inc** in our overridden **makeBigger()** method, but this negates much of the point of inheritance, and could become very tedious and potentially error-prone if we were overriding a more complicated method comprising many statements. A better way to handle it is to use the **inherited** keyword; **inherited** does whatever the method (or property) we're overriding would have done if we hadn't just overridden it. So, using inherited, a better way to define **greenBlob** would be:

```
greenBlob: Blob
      bulk = 3
      name = ('green ' + inherited)
      makeBigger(inc)
      {
         inherited(inc);
         "\^<<theName>> just got bigger! ";
      }
;
```

There are a couple of things to note here. The first (to reiterate a point made previously) is that when we use the **inherited** keyword in a method, we must use it with the same argument list as the method we're overriding; though not necessarily with the same argument list as the method we're defining: the following would be legal:

```
      makeBigger()
      {
         inherited(2);
         "\^<<theName>> just got bigger! ";
      }
```

The other thing to note is that we can also use the **inherited** keyword to retrieve the value of an inherited property (in this case the name 'blob' from **Blob**). Note also the syntax we employed here, setting the name property of **greenBlob** to an expression in brackets. This is *exactly* equivalent to writing:

```
      name  { return 'green ' + inherited; }
```

A further advantage of using the **inherited** keyword in situations like these is that if we subsequently realize we want to make changes to the base class (in this case Blob), the changes will then automatically be carried through to all the classes and objects that inherit from it. Suppose, for example, that we later decide that all the Blobs in our game should be called gooey blobs, and that no Blob should be allowed to grow beyond a certain maximum bulk. We might then rewrite our definition of the Blob class thus:

```
class Blob: Thing
      bulk = 2
      weight = 2
      maxBulk = 10
      makeBigger(inc)
      {
            if(bulk + inc <= maxBulk)
                  bulk += inc;
            else
                  bulk = maxBulk;
      }
      makeLighter()
      {
            if(weight > 0)
                  weight-- ;
      }
      name = 'gooey blob'
;
```

Then the enforcement of a maximum bulk will now also apply to the **greenBlob** object, whose name will now automatically become 'green gooey blob'. Furthermore, when we spot the obvious bug (namely that we hadn't allowed for the possibility that the *inc* parameter to **makeBigger(inc)** might be a negative number), whatever fix we apply to the **Blob** class will automatically apply to the **greenBlob** object and to any other class or object derived from the **Blob** class – provided we've used the **inherited** keyword when overriding the **makeBigger()** method (of course, it's also perfectly all right *not* to use the **inherited** keyword when we want the overridden method to do something substantially different from its behaviour on the class we're inheriting from, so that the inherited behaviour is of no use to us).

In addition to overriding methods and properties, we can also modify classes (and objects). Suppose that instead of defining a new **Blob** class, what we really wanted to do was to add the **makeBigger()** functionality to the library's **Thing** class. We could do this quite straightforwardly by modifying **Thing**:

```
modify Thing
    maxBulk = 10
    minBulk = 0
    makeBigger(inc)
    {
      bulk += inc;
      if(bulk > maxBulk)
            bulk = maxBulk;
      if(bulk < minBulk)
            bulk = minBulk;
    }
;
```

What this actually does is rename existing **Thing** class to some strange internal name like **ae45** and then create a new **Thing** class which inherits everything from it apart from the bits we've changed or overridden. Anything defined to be of class **Thing** or as inheriting from **Thing** now uses our new **Thing** class.

Note that we can also use the **inherited** keyword in a modified class, and that it works just the same way as it does in a class or object definition; that is it does whatever the method we're inheriting from would have done if we hadn't overridden it. For example, suppose the **makeBigger()** method were defined on a modification of **Thing** in some extension we were using, and we wanted to make a further modification to make the **makeBigger()** method display a message. We could then do this:

```
modify Thing
      makeBigger(inc)
      {
            inherited(inc);
            if(inc != 0)
                  "\^<<theName>> just got <<inc > 0 ? 'bigger' : 'smaller'>>! ";
      }
;
```

This shows that we can modify the same class (or object) as many times as we like, in which case the modifications take effect in the same order as they appear in the source code (which is one major reason why the library files always need to come first in our build: we can't modify a library class before the library defines it!). It should also be noted that we can, of course, equally well use **inherited** to inherit the behaviour of a method (or property) defined in the library, e.g., to make every **Openable** object remember if it has ever been opened:

```
modify Openable
      hasBeenOpened = nil
      makeOpen(stat)
      {
            inherited(stat);
            if(stat)
                  hasBeenOpened = true;
      }
;
```

We sometimes need more control over where we inherit from. For example, suppose we want to define an object that behaves like a **Decoration** in just about every respect, except that we want a special response for touching it. By default, touching a **Decoration** will produce the standard "not important" message. No get a different response, we need to override **verifyDobjTouch()** (don't worry about the name or purpose of this method for now; we'll be covering that kind of thing in the next chapter). But we can't just use the **inherited** keyword by itself, since that will simply inherit **Decoration**'s handling, which is what we want to change. What we actually want is **Thing's** version of **verifyDobjTouch()**. We can get that by using inherited plus the name of the class we want to inherit from:

```
hangingRug: Decoration 'hanging faded tatty rug*rugs' 'hanging rug'
   "It looks quite faded, and more than a little tatty. "
    feelDesc = "It feels quite rough. "
    verifyDobjTouch() {  inherited Thing; }
;
```

Note, however, that we can only do this with a class that the object (or class) we're defining actually inherits from at some point (however indirectly). If we want to borrow a method (or property) from some class that's nowhere in the inheritance tree of the class or object we're defining, we can use the **delegated** keyword instead, for example:

```
weather: Topic 'weather'
    name = 'weather'
    theName = ( delegated Thing )
    theNameFrom(str) { return delegated Thing(str); }
;
```

The **Topic** class (which we'll encounter again later) inherits from the **VocabObject** class, but not from the **Thing** class, which defines **theName** and **theNameFrom()**. **Thing.theName** is defined as **(theNameFrom(name))**, so if (for whatever obscure reason) we want **weather.theName** to evaluate to 'the weather', we need to borrow both the **theName** and **theNameFrom()** methods from **Thing**. The above example illustrates how we can do this using the **delegated** keyword.

The `modify` keyword lets us change an object or class definition, but only within certain limits. In particular it doesn't let us change the superclass list of the object or class we're modifying. For example, if we use `modify` to change the behaviour of the `OpenableContainer` class the one thing we can't modify is the fact that it inherits from the classes `Openable` and `Container`. If we want to start completely from scratch with the definition of an object that's been previously defined, we can do so using the `replace` keyword. For example, the following would be possible (though not particularly useful):

```
replace OpenableContainer: Thing
   verifyDobjOpen() { illogical('Just because this looks openable doesn\'t mean
      I'm going to let you open it! '); }
;
```

Following the `replace` keyword we go on to define the class (or object) just as if we were defining it completely from scratch. The `replace` keyword can also be use to replace functions that have been previously defined.

For more information about the topics covered in this excursus, read the relevant parts of the articles 'The Object Inheritance Model', 'Object Definitions', 'Expressions and Operators' and 'Procedural Code' in the *TADS 3 System Manual*.

5.3 In, On, Under, Behind

5.3.1 Kinds of Container

After that somewhat lengthy (but nevertheless important) excursus, we can return to the main topic of this chapter, namely containment. We have already seen that we can use the `Container` class to put things in; we should now look at some related classes. First, here is the list of classes for things than can contain other things within them:

- `BasicContainer` – This is the base type for the other Container types; it has no action handling, which means that the player can't put things in a `BasicContainer`, but things can be put inside by game code. We seldom use this class in a game.
- `Container` – the standard container type we've met already. This is a straightforward container like a bin, bag or sack that we can put things in.
- `Dispenser` – a container for a special type of item, often one where we can take an item from the `Dispenser` but not return it (like a box of matches or a roll of paper towels). The `canReturnItem` property (`nil` by default) determines whether or not an items can be returned to the `Dispenser`, and the `myItemClass` property (`Dispensable` by default) contains the class of objects dispensed by (and possibly accepted by) the `Dispenser`. We'll meet a particular example of this (the `Matchbook`) in the chapter on Darkness and Light.

- **`RestrictedContainer`** – a container that can contain only certain objects (for example, only batteries can go in a flashlight). We can either list the objects that can fit a **`RestrictedContainer`** in its **`validContents`** property, or override the **`canPutIn(obj)`** method to determine whether *obj* may be put in the **`RestrictedContainer`**; the method should return **`true`** to allow insertion or **`nil`** to forbid it. (**`RestrictedContainer`** is a particular type of **`RestrictedHolder`**).
- **`SingleContainer`** – a container that can only hold one object at a time; putting an object in a **`SingleContainer`** that already contains something causes that thing to be removed from the **`SingleContainer`** first.
- **`StretchyContainer`** – a container that grows in bulk to match the bulk of its contents (a full sack is bulkier than an empty sack, for instance). Even an empty sack can may have some bulk, however, so we can use the **`minBulk`** property to define the bulk of the **`StretchyContainer`** when empty. The **`bulk`** property gives the intrinsic bulk of the empty **`StretchyContainer`**; use the **`getBulk()`** method to report its distended bulk (i.e. the total bulk allowing for the expansion due to its contents).
- **`OpenableContainer`** – a container that can be opened or closed, and usually hides its contents when closed (see further on materials below). As with doors we can use the **`isOpen`** property to test whether an **`OpenableContainer`** is open or closed, but should use the **`makeOpen(stat)`** method to open or close it under programmatic control. If we want an **`OpenableContainer`** to start out open, set its **`initiallyOpen`** property to true.
- **`LockableContainer`** – a kind of **`OpenableContainer`** that can be locked or unlocked. Note, however, that a **`LockableContainer`** doesn't need a key; a **`LockableContainer`** models a container that can be locked or unlocked with some kind of catch. There's little obstacle to a player opening a **`LockableContainer`**, since the default library behaviour is to unlock it via an implicit action. The **`initiallyLocked`** property (true by default) defines whether the container starts out locked. Use the **`makeLocked(stat)`** method to lock or unlock a **`LockableContainer`** under program control.
- **`KeyedContainer`** – a kind of **`LockableContainer`** that needs a key to unlock it. We'll discuss this further when we come to the chapter on locks and keys.
- **`BagOfHolding`** – this is not a class of container, but a mix-in class for use with a container class. If the player character has a limited **`bulkCapacity`** and his/her hands become too full but the player character is carrying a BagOfHolding, items will be moved from his/her inventory to the BagOfHolding to make room for the player character to take something else. The **`affinityFor(obj)`** method can be used to return a value defining how willing a given BagOfHolding is to have *obj* moved to it; the return values should range between 0 (*obj* can't be

moved to this BagOfHolding at all) to 200 (this object or type of object is what this BagOfHolding is particularly meant for). The default value is 100.

The use of these various classes shouldn’t present any particular problems, but a couple of examples may be helpful here:

```
+ hole: RestrictedContainer, Fixture 'small round hole*holes' 'small round hole'
     validContents = [roundPeg, pen]
     notifyInsert(obj, newCont)
     {
          "As you insert <<obj.theName>> in the hole, you hear a click come
           from the door. ";
          blackDoor.makeLocked(nil);
     }
     notifyRemove(obj)
     {
         "There's a click from the black door as you remove <<obj.theName>>. ";
         blackDoor.makeLocked(true);
     }
     bulkCapacity = 1
;
```

Here we have a special device for unlocking a door, a round hole that unlocks a door when something is inserted and locks it again when the object is removed. Since it’s a small hole, only small objects can fit in, and only one at a time (`bulkCapacity = 1` should ensure this). Because of the shape of the hole, only the round peg and the pen will actually fit. Note that we make the hole a `Fixture` as well as a `RestrictedContainer`; by default containers are portable, to stop them being moved around we have to add a `NonPortable` class to their class list.

A second example:

```
+ briefcase: LockableContainer 'large leather briefcase/case' 'briefcase'
    "It's quite large, and made of leather. "
;

++ document: Readable 'document*documents' 'document'
    "It's marked <FONT COLOR=RED>TOP SECRET</FONT> at the top. The
    rest seems to be in code. "
    readDesc()
    {
       if(codeBook.seen)
          "It looks like the secret plans for a new IF language that will
               revolutionize the production of Interactive Fiction! ";
       else
          "It's in code; you won't be able to decipher it until you find
           the key. ";
    }
;
```

Here the briefcase is portable, and has a lock, but the lock is a simple catch that can be unlocked without a key. Inside is briefcase is a document that will be found once the briefcase is open, but which won't be visible while the briefcase is closed.

5.3.2 Container Materials

In the previous example the document is effectively invisible while it's inside the closed briefcase; that's reasonable, we can't see through leather. But some containers such as glass jars or wire cages, may be made of materials we can see through. We can cater for this in TADS 3 with the **material** property.

The TADS 3 library defines the following kinds of material:

- **adventium** – opaque to all senses (sight, sound, smell and touch).
- **glass** – transparent to light, but opaque to the other three senses (i.e. we can see through it, but not hear, smell or touch through it).
- **paper** – opaque to touch and sight, but allows smells and sounds through.
- **fineMesh** – transparent to all senses except touch.
- **coarseMesh** – transparent to all senses, but doesn't allow objects to pass through.

If we needed some other combination of senses, it would be easy enough to define our own material. For example, suppose we wanted a material that allowed only sound to pass, we could define new material (let's call it cardboard) like this:

```
cardboard: Material
    seeThru = opaque
    hearThru = transparent
    smellThru = opaque
    touchThru = opaque
;
```

To make use of these materials, we simply need to assign them to the **material** property of the container in question. For example, suppose we want a glass jar containing a pebble and a small cage containing a canary:

```
+ glassGar: OpenableContainer 'glass jar*jars'  'glass jar'
    material = glass
;

++ pebble: Thing 'small round pebble/stone*pebbles' 'pebble'
    "It's small and round. "
;

+ cage: OpenableContainer '(bird) birdcage/cage' 'birdcage'
   material = fineMesh
;

++ canary: Actor 'canary/bird*birds' 'canary'
;
```

Note that there's really only any point in doing this with an openable container, since unless the container can be closed, its contents will be visible (and tangible, audible and smellable) in any case. As yet we haven't really said much about sound, smell and touch, so of the materials we've talked about, only glass and adventium are likely to be immediately useful. We'll deal with the other senses in a later chapter.

5.3.3 Other Kinds of Containment

So far we've concentrated on containers we can put things *in.* But it's also common in Interactive Fiction to have things we can put things *on*: tables, desks, trays and things like that. For this we use the **Surface** class. As with containers, Surfaces are portable unless we make them otherwise by mixing them in with a **NonPortable** class. So, for example, we might have:

```
+ table: Surface, Heavy 'table*tables'  'table'
;

++ tray: Surface 'tray*trays' 'tray'
;

+++ mat: Surface 'mat*mats' 'mat'
;

++++ bowl: Container 'bowl*bowls' 'bowl'
;

+++++ grape: Food 'grape*grapes*fruit' 'grape'
;
```

In this example the grape is in a bowl which is resting on a mat which is resting on a tray which is resting on the table. The table can't be moved (because it's too heavy), but the tray and the mat can both be taken (as, of course, can the bowl and the grape).

RestrictedSurface works just like **RestrictedContainer** (both inherit from **RestrictedHolder**, which is where the restricting behaviour is mostly defined). At first sight the idea of a **Surface** which will only allow certain objects (or certain kinds of object) to be put on it may seem a little strange, but the point becomes clearer if we think of a **RestrictedSurface** not primarily as a flat surface like a table-top, but simply as something we might reasonably try to put only certain kinds of thing on, such as a peg, coat-hanger or coat-stand, so we might have something like:

```
+ hanger: RestrictedSurface 'wire coat-hanger/hanger*hangers' 'coat-hanger'
    validContents   = [whiteShirt, blueShirt, brownCoat]
;

+ peg: RestrictedSurface, Fixture 'wooden peg*pegs' 'peg'
   validContents = [floppyHat, brownCoat]
;
```

Putting things in and on other things is pretty common in IF. Less common, but still useful, is putting things under or behind other things. For this TADS 3 defines the following classes:

- **Underside** – something we can put things under.

- RearContainer – something we can put things behind.
- RearSurface – essentially the same as a RearContainer but models the contents as being attached to the back of the object rather than merely sitting behind it. The only practical difference between a RearContainer and a RearSurface is that moving a RearSurface moves its contents along with it, whereas moving a RearContainer abandons the contents, leaving them behind where the RearContainer used to be.

There are also Restricted versions of each of these classes (RestrictedUnderside, RestrictedRearContainer and RestrictedRearSurface) which restrict their contents in much the same way as RestrictedContainer and RestrictedSurface. Their use is unlikely to be very common, but one can imagine, for example, that we might want to use a RestrictedRearContainer to model a very narrow space behind a piece of heavy furniture, into which only a sheet of paper might fit.

The Underside, RearContainer and RearSurface classes particularly lend themselves to be used in conjunction with the Hidden class, since it could well be that the player won't notice what's under or behind something until s/he looks under or behind it; for example:

```
+ rug: Underside 'small dark rug*rugs'  'rug'
;

++ silverCoin: Hidden 'silver coin*coins' 'silver coin'
;

+ mirror: RearSurface 'square mirror*mirrors' 'mirror'
    initSpecialDesc = "A square mirror is hanging on the wall. "
;

++ bankNote: Hidden 'bank note/banknote*notes banknotes' 'banknote'
;
```

There's a further point to note about this example. If the player takes the rug, the silver coin will be revealed in any case (because it's assumed to have been left lying on the floor). If the player takes the mirror, the banknote isn't automatically revealed because it's assumed to be stuck to the back of the mirror (and won't be found until the player enters the command **look behind mirror**). If the mirror had been a RearContainer instead of a RearSurface, it would have behaved more like an Underside; that is, moving it would have revealed the banknote that would then have been notionally left behind on the wall (or perhaps notionally have fallen to the floor). This behaviour is defined on the SpaceOverlay class, from which both RearSurface and Underside inherit. Look up SpaceOverlay in the *Library Reference Manual* for further details (including how its behaviour can be tweaked).

We have now met four types of containment: in, on, under and behind. Thing, and hence all these classes that descend from Thing, provide the property objInPrep, which defines the preposition to be used for objects located within. Thus `Thing.objInPrep` is 'in' (and `BasicContainer` inherits the same value from `Thing`), `Surface.objInPrep` is 'on', `Underside.objInPrep` is 'under' and `RearContainer.objInPrep` is 'behind'. This property can be used to tweak certain kinds of message describing the whereabouts of an object. It is used indirectly, in that `actorInPrep` is used to build phrases describing the whereabouts of an actor (something we'll come back to in a later chapter) and `actorInPrep` by default takes its value from `objInPrep`. We can also make things be listed as 'beneath' something rather than 'under' something, say, simply by changing `Underside.objInPrep` to 'beneath'.

If we want more control over how objects in, on, under or behind other things are listed (more than simply changing the preposition, that is), we need to override the appropriate `Lister`. The particular listers to be used with a particular class of `Container`, `Surface`, `Underside` or `RearContainer` are defined on various properties of that class. A fuller explanation is beyond the scope of this manual, both because it would be quite lengthy, and also because it's already provided elsewhere: for the full story see the article on 'Lists and Listers' in the *TADS 3 Technical Manual* (but you don't have to rush to do this straight away; it can wait until you need it).

Finally, although we have now seen four types of containment (in, on, under and behind), apart from the minor differences between them that we have noted, they all basically use the same containment mechanism. That is the containing object (whether a `Container`, `Surface`, `Underside` or `RearContainer`) maintains a list of the things it contains (in, on, under or behind) in its `contents` property, and the contained objects keep a note of what they're contained by in their `location` property. This means that a given object can support only one kind of containment relation. If it's a `Container`, we can put things in it, but not on it. If it's a `Surface`, we can put things on it, but not in it, under it or behind it. For some things that's okay, but for others it's an unrealistic restriction. We can often put things under a bed or table as well as on top of it. A desk next to a wall might have things on it, under it, in it and behind it. At first sight the TADS 3 world model seems not to allow for this. It turns out that TADS 3 does provide a means of dealing with this kind of situation, but before we go on to look at it, we first need to take a closer look an anonymous objects.

5.4 Coding Excursus 8 – Anonymous and Nested Objects

Hitherto, we've given every object a name when we've defined it (here 'name' refers to the object identifier that comes before the class list, not to the `name` property). For example, suppose the room description mentioned faded pink wallpaper, so we decided to implement the wallpaper as a `Decoration`:

```
+ wallPaper: Decoration 'faded pink wallpaper' 'wallpaper'
    "It looks like the kind of thing you'd associate with a Victorian nursery;
     it's almost faded enough to be that old. "
```

```
;
```

The only real function of this object is to provide a response to **examine wallpaper** that doesn't deny the wallpaper's existence. We'll never need to refer to the wallpaper object in any other piece of code. In such an instance there's actually no need to give the wallpaper object an identifying name, we can instead declare it as an *anonymous* object:

```
+ Decoration 'faded pink wallpaper' 'wallpaper'
    "It looks like the kind of thing you'd associate with a Victorian nursery;
     it's almost faded enough to be that old. "
;
```

Although this kind of anonymous object declaration is particularly useful with decoration-type objects, it's by no means restricted to them; we can use it for absolutely any object that we don't need to refer to by its identifying name elsewhere. This can make our code a bit more compact, and also spares us the trouble of having to think up lots of identifying names for unimportant objects. In any case, if we declare an anonymous object and later find that we do need to refer to it in some other part of code, we can always go back and give it an identifying name.

Another use of anonymous objects is as *nested* objects. A nested object (which is necessarily anonymous) is one that is defined directly on the property of another object. We have already seen examples of this in defining various kinds of `TravelConnector` on the directional properties of rooms, e.g.:

```
meadow: OutdoorRoom 'meadow'
    "The ground becomes distinctly marshier to the north. "
    north: TravelMessage { ->marsh  "You step cautiously into the marsh. " }
;
```

We should note several points about this kind of definition.

First, we can always *refer* to a nested object using the name and relevant property of the enclosing object; in this instance `meadow.north` will give us a reference to the `TravelMessage` object. But the nested object remains anonymous; meadow.north is the name of the north property of the meadow, which just happens to contain a `TravelMessage` object right now (but which in principle could later be changed to contain something else, even if we're unlikely ever to change it in practice); it's not the name of the `TravelMessage` object.

Second, when defining a nested object, we use exactly the same syntax (following the colon) as we would for defining an ordinary object, starting with the class list, except that we can't give it a name and that we must use the brace notation (`{ }`) to delimit the object definition.

Third, when defining a nested object using a template, we can either put the bits belonging to the template inside the braces (as in the above example) or between the class list and the opening brace (as is also the case when defining an ordinary object with the brace notation).

The previous example, purely using a template, may slightly obscure the fact that a nested object can be defined with properties and methods just like any ordinary object, for example:

```
desk: Surface, Heavy 'desk*desks furniture' 'desk'
     underDesk: Underside
     {
          name = 'desk'
          bulkCapacity = 5
          notifyRemove(obj)
          {
               "You pull <<obj.theName>> out from under the desk. ";
          }
     }
;
```

It's often useful for a nested object's methods and properties to refer to its enclosing object. For this purpose we can use the special property `lexicalParent`. For example, we could slightly amend our first example to:

```
meadow: OutdoorRoom 'meadow'
    "The ground becomes distinctly marshier to the north. "
    north: TravelMessage { ->marsh  "You step cautiously from
        <<lexicalParent.theName>> into the marsh. " }
;
```

Going north from the meadow to the marsh would then result in the display of the message "You step cautiously from the meadow into the marsh."

Note that it is very easy to forget to use `lexicalParent` *to refer to the enclosing object when working with nested objects. This is a very common potential source of bugs!*

For further information on anonymous and nested objects see the 'Object Definitions' article in the *TADS 3 System Manual*.

5.5 Complex Containers

We left our discussion of containers at the point of talking about how to implement objects that need more than one kind of containment, for example a table we can put things both on and under, or a floor-standing cabinet one can put things both on and inside. The TADS 3 solution is the `ComplexContainer` class. This works by incorporating a number of sub-objects, each representing one kind of containment. Each of these sub-objects is defined on the appropriate property of the `ComplexContainer`, and needs to be defined as belonging to the `ComplexComponent` class together with whatever class is appropriate to its containment type. For example, to implement a cabinet we can put things in, on, under or behind we could do the following:

```
+ cabinet: ComplexContainer 'cabinet*cabinets' 'cabinet'
    subSurface: ComplexComponent, Surface { }
    subContainer: ComplexComponent, OpenableContainer
    {
        bulkCapacity = 10
    }
    subRear: ComplexComponent, RearContainer { }
    subUnderside: ComplexComponent, Underside { bulkCapacity = 10 }
;
```

From the player's point of view, this will appear to be a cabinet that the player can put things in, on, under or behind. What actually happens is that there are five objects: the cabinet itself, and four `ComplexComponents` representing the spaces in, on, under and behind the cabinet. Each of these `ComplexComponents` automatically takes its name from its `lexicalParent`, the cabinet, so that objects within these `ComplexComponents` are described as being in, on, under or behind the cabinet. None of these `ComplexComponents` has any `vocabWords`, so any commands targeted at the cabinet will be fielded by the `cabinet` object; but since the cabinet is a `ComplexContainer`, it automatically redirects certain actions to the objects defined on its `subXXXX` properties, provided they're present. For example, **open, close, lock, unlock, put in** and **look in** are all directed to the `subContainer`; **put on** is redirected to the `subSurface`; **put under** and **look under** are redirected to the `subUnderside`; and **look behind** and **put behind** to the `subRear` (provided objects have been defined on the relevant properties).

We may often want some objects to start out in one or other part of a `ComplexContainer`. For example we might want a piece of paper to have slipped down behind the cabinet, an ornamental vase to be on top of the cabinet, a mat to be inside the cabinet, and a coin to be on the floor under the cabinet. One way of doing this would be to define the location property of each of these explicitly:

```
vase: Container 'vase*vases' 'vase'
    location = cabinet.subSurface
;

paper: Readable, Hidden 'piece/paper'  'piece of paper'
   location = cabinet.subRear
;

mat: Surface 'mat*mats'  'mat'
   location = cabinet.subContainer
;

coin: Hidden 'coin*coins' 'coin'
    location = cabinet.subUnderside
;
```

But there is another way of doing this, which may often be more convenient. We can instead use the + notation in the normal way, and use a special notation involving the

subLocation property and a property pointer, which is a property name preceded by an ampersand (&); this gives a reference to the property rather than the value of the property, a way of saying "we want to note that we want to do something with this property but we don't want to evaluate it just yet". The above example would then become:

```
+ cabinet: ComplexContainer 'cabinet*cabinets' 'cabinet'
    subSurface: ComplexComponent, Surface { }
    subContainer: ComplexComponent, OpenableContainer
    {
        bulkCapacity = 10
    }
    subRear: ComplexComponent, RearContainer { }
    subUnderside: ComplexComponent, Underside { bulkCapacity = 10 }
;

++ vase: Container 'vase*vases' 'vase'
    subLocation = &subSurface
;

++ paper: Readable, Hidden 'piece/paper'  'piece of paper'
    subLocation = &subRear
;

++ mat: Surface 'mat*mats'  'mat'
    subLocation = &subContainer
;

++ coin: Hidden 'coin*coins' 'coin'
    subLocation = &subUnderside
;
```

While this doesn't save a huge amount of typing, it does make it easier to associate objects contained in an **ComplexContainer** with that **ComplexContainer** in the way we lay out the code.

Note that we don't need to define all four **subXXXX** properties on a **ComplexContainer**; we simply define whatever combination we want. So if all we want is a table we can put things on and under and a washing machine we can put things on and in, we'd define:

```
+ table: ComplexContainer, Heavy 'kitchen table*tables'  'table'
     subSurface: ComplexComponent, Surface { }
     subUnderside: ComplexComponent, Underside { }
;

+ washingMachine: ComplexContainer, Heavy 'washing machine' 'washing machine'
    subSurface: ComplexComponent, Surface { }
    subContainer: ComplexComponent, OpenableContainer
    {
       notifyInsert(obj, cont)
       {
           if(!obj.ofKind(Wearable))
           {
                "You're only meant to put clothes in there! ";
                exit;
           }
```

```
        }
        bulkCapacity = 15
    }
;
```

Note too that it's sometimes necessary to use a ComplexContainer when at first sight it looks as if an OpenableContainer should do the job. This is normally the case whenever we want to give a container any components, since its components are considered to be *inside* the container, and so will disappear from scope when the container is closed. For example, suppose we wanted a briefcase with a handle and a combination lock; we might (erroneously) try something like this:

```
briefcase: LockableContainer 'brown briefcase/case' 'briefcase'
     "It's a light brown case with a handle and combination lock. "
;

+ Component 'handle*handles' 'handle' // DON'T DO THIS!
;

+ Component 'combination lock*locks'  'lock' // OR THIS!
;
```

The problem with this is that both the handle and the lock start out *inside* the briefcase, so the player can't interact with them when the briefcase is closed (which probably isn't what we want at all!). Even worse, once this code is developed a little further to make the combination lock the mechanism for unlocking the case, it'll become impossible to unlock it, since the combination lock will be locked inside the very case it's meant to unlock.

The way round this kind of situation is to make the container a ComplexContainer; we should start out with something like this:

```
briefcase: ComplexContainer 'brown briefcase/case' 'briefcase'
     "It's a light brown case with a handle and combination lock. "
     subContainer: ComplexComponent, LockableContainer { }
;

+ Component 'handle*handles' 'handle'
;

+ Component 'combination lock*locks'  'lock'
;
```

This will then work as intended, since the handle and the lock aren't in the subContainer; they'll now appear effectively on the outside of the briefcase.

Another case where we'd need to use a ComplexContainer to model an openable container is where we want to implement the container's door as a separate object. For this purpose we can use the special ContainerDoor class, like this:

```
cupboard: ComplexContainer 'cupboard'  'cupboard'
     subContainer: ComplexComponent, OpenableContainer {  }
```

```
;

+ ContainerDoor, Component  '(cupboard) door*doors' 'cupboard door'
;

+ cheese: Food  'piece/cheese'  'piece of cheese'
    subLocation = &subContainer
;
```

This puts the piece of cheese inside the cupboard, and the cupboard door on the outside of the cupboard, where it belongs.

Exercise 12: One place where you might expect to find quite a few containers of different types in a kitchen, so try implementing one now. Your kitchen should include a work top (fixed in place, of course), on which is a cookery book hiding a note underneath, an apron hanging from a peg, a box full of cutlery lying in the corner, a cooker (with a door), you can put things in, on or behind; there's a cake in the oven, and the instruction leaflet for the cooker has fallen down behind. On the cooker (or stove) is a pot and a saucepan with a handle. The kitchen has a table you can put things on or under, and under it is a red box containing a can opener (or tin opener). Fastened to the wall is a cabinet containing a glass jar with a number of sugar cubes in it. There's also a soup can in it, but the full implementation of that may have to wait. The kitchen also has a roll of paper towels you can take from the roll one at a time (but obviously can't return to the roll). On the wall is a clock with a manufacturer's label stuck on its back. When you've got as far as you can, compare your version with the Containers example. To complete this exercise, you'll need material from the following chapter.

6 Actions

6.1 Taxonomy of Actions

Although there is quite some way to go to cover all the main features of TADS 3, we've now covered the fundamentals on the TADS 3 world model. But that doesn't enable us to write any very interesting games; we can build a map and populate it with objects, but we can't make them *do* much; indeed, as yet, we can't make them do anything beyond their default behaviour. We can build a very basic simulation, but we can't make a game. What makes a work of Interactive Fiction interesting is the way it responds to the player's commands, and in particular, the way its responses go beyond the basic library model. A great deal of the programming in IF consists in defining the response to player's commands. This chapter will lay the groundwork for doing this. We'll leave some of the finer details to a later chapter.

Before we can start coding responses to actions, we need to understand the types and parts of an action. In form, commands in TADS 3 (and most IF in general) take one of three forms:

- *verb* – for example **look**
- *verb direct-object* – for example **take ball**
- *verb direct-object preposition indirect-object* – for example **put ball in box**

In these commands, the *verb* part is always a verb in the imperative mood (the kind of verb form we use for giving orders), for example **go**, **take**, **put**, **drop**. The *direct-object* is generally a noun naming a game object; the direct object is the object on which we want the command to act on directly (taking, dropping or moving the ball, for example). Where the command involved two objects, the second object is called the *indirect object*; for example the box is the indirect object in the command **put ball in box**. The preposition is the word between the two objects defining how the indirect object is involved in the command (e.g. **put the ball *in* the box**, **cut the string *with* the knife**, or **give the book *to* the man**).

Sometimes, both English and TADS 3 allowed a variant form in which the indirect object comes before the direct object and the preposition (normally 'to') is omitted; for example **give Bob the ball** means **give the ball to Bob**; just as **throw Bob the ball** means **throw the ball to Bob**. With commands of this kind we have to translate the phrasing back to the longer form (including a preposition) to work out which is direct object, and which is the indirect object.

In working with actions in TADS we'll often see names (often of macros, which we'll explain shortly) containing 'dobj' and 'iobj' somewhere. It helps to recognize that these are nearly always abbreviations for direct object and indirect object.

The three kinds of actions we've encountered so far correspond to three classes of action in TADS 3:

- `IAction` - actions with a command only (and no objects), like **look.**
- `TAction` – actions with one object, the direct object, like **take ball**
- `TIAction` – actions with two objects, a direct object and an indirect object, like **put ball under table**.

It's often to useful to know what the current action is, who's carrying it out, and what objects are involved in it. For those purposes we can use the following pseudo-global variables (actually macros):

- `gAction` – the current action.
- `gActor` – the actor performing the current action.
- `gDobj` – the direct object of the current action.
- `gIobj` – the indirect object of the current action.

There's also a pair of macros (which look like functions) we can use to test what the current action is:

- `gActionIs(Something)` – returns `true` if the current action is SomethingAction.
- `gActionIn(Something, SomethingElse... YetSomethingElse)` – returns `true` if the current action is one of those listed.

These might be used like this:

```
     if(gActionIs(Take))
          "Don't be greedy – you're carrying quite enough already. ";
     if(gActionIn(PutIn, PutOn, PutUnder, PutBehind))
          "Just leave things where they are! ";
```

For a complete list of actions, go to the *Library Reference Manual*, and then click the *Actions* link third along from the right. A list of actions defined in the TADS 3 library will then appear in the bottom left-hand panel.

Amongst the actions listed are a number that look a bit like `TActions` or `TIActions`, but are in fact something a bit different. Examples of such actions include:

- GO NORTH
- PUSH THE TROLLEY EAST
- TYPE SUGARPOP ON TERMINAL
- ASK BOB ABOUT THE WEATHER
- LOOK UP RABIES IN MEDICAL TEXTBOOK

In the first of these NORTH is not the direct object of the GO command; this is an `IAction`, not a `TIAction`. In TADS 3 north is a direction, not a Thing (grammatically it's more like an adverb than a noun in this context). Indeed, an IF player will normally abbreviate this kind of command to just the direction, **north** or **n**. Similarly, the second command is not a `TIAction`, but a `TAction`; TROLLEY is the direct object but there is no indirect object (and in particular, EAST is not the indirect object). In the command TYPE SUGARPOP ON TERMINAL, it may look as if SUGARPOP is the direct object and TERMINAL the indirect object, but this is not so: SUGARPOP is not the name of an object in the game, but a string of characters (perhaps a password) that the player wants to type on the terminal. Thus this is not a `TIAction` (as it might first appear) but a `LiteralTAction`, in which the terminal is the direct action (accessible as `gDobj`) and SUGARPOP is the 'literal phrase' (accessible as `gLiteral`). Neither of the final two examples is a `TIAction` either (despite initial appearances); the way TADS 3 defines these two actions (and others like them), THE WEATHER and RABIES are topics, not things (since the player is not restricted to talking about things implemented in the game). ASK ABOUT and LOOK UP are `TopicTActions`.

The difference between a topic and a literal may not be immediately apparent. The difference is that a Literal is simply a piece of text, with no reference to any simulation object in the game. A topic may just match a piece of text, but it may also refer to a `Topic` object or a simulation object (we'll talk more about `Topic` objects in the next chapter). If the command had been ASK BOB ABOUT SUSAN, the action would still have been a `TopicTAction` (with Bob as the direct object), even if there was an actor called Susan implemented in the game; but even though SUSAN would, in the first instance, be matched as an topic, a connection could also be made between this topic and the Susan object (don't worry if this explanation seems a little obscure right now, we'll unpack it further in later chapters).

The main point to note right now is that there are four more types of action:

- `LiteralAction –` a command consisting of a *verb* plus some *literal text*.
- `LiteralTAction` – a command consisting of a *verb*, one *thing* (the direct object), and some *literal text* (e.g. **write foo on paper**).
- `TopicAction` – a command consisting of a *verb* plus one *topic* (e.g. **talk about the weather**)
- `TopicTAction` – a command consisting of a *verb* plus one *thing* (the direct object) plus one *topic* (e.g. **tell bob about the weather**).

Knowing the types of action is only the prelude to learning how to customize them, but before we go on to that, there's another couple of coding constructs it will be helpful to know about.

6.2 Coding Excursus 9 – Macros and Propertysets

6.2.1 Macros

Several times now we've referred to things called *macros* without really explaining what they are. Put simply, a macro is a kind of convenient abbreviation. Put a bit more technically, a macro is a piece of text that the *preprocessor* replaces with a predefined expansion before the compiler gets to work on the source file. For example, in reality TADS 3 has no global variables. Things that look like global variables are in reality the properties on some object (such a `libGlobal`). The current action, for example, is in reality `libGlobal.curAction`, but the macro `gAction` is defined as a convenient abbreviation for this. The current direct object is in reality `libGlobal.curAction.getDobj()`, but it's much easier just to be able to write `gDobj`.

Macros are defined using the keyword `#define`. The macros just mentioned are defined like this:

```
#define gAction (libGlobal.curAction)
#define gDobj (gAction.getDobj())
```

In effect, these are instructions to the preprocessor (which runs just prior to compilation), telling it that every time it sees the text `gAction` in the source file it should replace it with `(libGlobal.curAction)`, and that every time it sees the text `gDobj` in the source file it should replace it with `(gAction.getDobj())`. Note that this replacement is cumulative; having replaced `gDobj` with `(gAction.getDobj())` the preprocessor will replace `gAction` with `(libGlobal.curAction)` so that the full expansion of `gDobj` becomes `((libGlobal.curAction).getDobj())`; thus whenever we write `gDobj` in our source code, `((libGlobal.curAction).getDobj())` is what the compiler 'sees'. (Macro replacement is not, however, recursive; if a macro contains its own name in its expansion, it will not recursively expand its own name on any second or subsequent pass).

Note also that macros only take effect in the source file in which they are defined. If we want macros to take effect in several (or all) of our source files, we need to define them in a header file (one with a .h extension) and then include the header file in all our source files. This is one of the reasons why we need to put the following near the top of all our TADS 3 game source files:

```
#include <adv3.h>
#include <en_us.h>
```

This ensures that we can use all the macros defined in the TADS 3 library. If we defined some macros of our own we wanted to use in our own game, we might put them in a file called myGame.h then ensure we added the following near the top of all our source files:

```
#include "myGame.h"
```

We would probably use quote marks ("") rather than angle brackets (<>) here because we'd presumably put myGame.h in the same directory as all the other source files for our game.

Macros can be both simpler and more complicated than those we've seen so far. The very simplest form of macro just defines that the macro has defined; e.g.:

```
#define ExtraHandsome
```

This can then be used to define an optional block of code that's only compiled if the `ExtraHandsome` macro has been defined:

```
#ifdef ExtraHandsome
    modify me
         desc = "So unbelievably handsome you can't bear to look at yourself. "
   ;
#endif
```

Almost as simple is a macro that just gives a symbolic name to a constant:

```
#define  SpecialOptionCount 12
```

Rather more complicated is the function-type macro, which takes one or more argument, for example:

```
#define Double(X)  (X * 2)
```

This looks a bit like a function, but what actually happen is that the preprocessor substitutes whatever value we put in for X and replaces it with that value in the expansion. For example, if the preprocessor encounters `Double(3)` it will replace it with `(3 * 2)` before the compiler gets to work on it.

Function-type macros can also use *token pasting* to construct a programming token out of its arguments, using the `##` token pasting symbol. This is best explained by means of an example. Suppose we define the following macro:

```
#define gActionIs(action)  (gAction.actionOfKind(action##Action))
```

Then when the processor comes across `gActionIs(Take)` it will replace it with `(gAction.actionOfKind(TakeAction))` (this is actually a slightly simplified version of the `gActionIs()` macro defined in the library).

The foregoing is only a quick sketch of what macros can do. To get the full story on macros, as well as including header files and other features of the preprocessor see the article on 'The Preprocessor' in Part III of the *TADS 3 System Manual*.

To find out what macros the library defines and what they do, click the *Macros* link in the bar at the top of the *Library Reference Manual*. A list of library macros will then appear in the bottom left-hand panel. You can scroll through this list and click on any macro you're interested in to see its definition, often along with a brief description of what it's for.

6.2.2 Propertysets

A *Property Set* is simply a short-cut way of defining a number of related properties with similar names. The **propertyset** keyword is used to define the pattern to be used in such a set of properties. This pattern uses an asterisk (*****) as a placeholder for the variable part of the property name. For example, suppose we wanted to define a whole set of properties that included 'put' in their name; we might define:

```
propertyset 'put*'
{
     In(x)   {  moveInto(x); }
     On() { "You can't do that. " }
     Under(x) { "There's no room under <<x.theName>>. "}
     Behind(x} { }
     Msg = 'You put it somewhere. '
}
```

This is exactly the same as defining:

```
PutIn(x)   {  moveInto(x); }
PutOn() { "You can't do that. " }
PutUnder(x) { "There's no room under <<x.theName>>. "}
PutBehind(x} { }
PutMsg = 'You put it somewhere. '
```

And this, indeed, is precisely what the compiler 'sees'.

A macro definition can be combined with a propertyset definition; for example the library defines:

```
dobjFor(action)  objFor(Dobj, action)
iobjFor(action) objFor(Iobj, action)
objFor(which, action) propertyset '*' ## #@which ## #@action
```

The effect of this somewhat arcane definition is as if we'd defined:

```
dobjFor(action)  propertyset '*Dobj' ## #action
iobjFor(action)  propertyset '*Iobj' ## #action
```

This is turn means that the following code (of a kind that's very common when we start customizing and defining actions):

dobjFor(Take)

```
{
    preCond = [touchObj]
    verify()
    {
        if(meetsObjHeld(gActor)
          illogicalAlready('You are already holding it! ');
    }
    check() { }
    action()
    {
       moveInto(gActor);
       "Taken. ";
   }
}
```

This is exactly equivalent to:

```
    preCondDobjTake = [touchObj]
    verifyDobjTake()
    {
        if(meetsObjHeld(gActor)
          illogicalAlready('You are already holding it! ');
    }
    checkDobjTake() { }
    actionDobjTake()
    {
       moveInto(gActor);
       "Taken. ";
    }
```

What's important here is not so much that we understand every step of the process by which the first piece of code becomes equivalent to the second, but that we recognize the equivalence.

For the full story on property sets, read the 'Property Sets' section of the 'Object Definitions' article in Part III of the *TADS 3 System Manual*.

6.3 Customizing Action Behaviour

When it comes to customizing the behaviour of existing actions (or defining the behaviour of new actions), actions basically divide into two kinds: those that have direct objects (TAction, TIAction, TopicTAction and LiteralTAction) and those without (IAction, TopicAction, and LiteralAction). The latter kind is easier to explain, so we'll start with that first.

6.3.1 Actions Without Objects

The behaviour of an action that has no objects is defined in the **`execAction()`** method of the action class. To customize the behaviour of an existing action, we simply modify the appropriate action class and override its **`execAction()`** method. For example, if we want to customize the way the Jump action works, we might do this:

```
modify JumpAction
    execAction()
    {
         if(gActor.getWeight > 15)
              "You're too weighed down to jump. ";
         else
              "You jump vigorously, but it does no good. ";
    }
;
```

If we're modifying the behaviour of an action defined in the library, it's a good idea first to look at how the library defines it, however. For example, the library defines **SleepAction** as:

```
DefineIAction(Sleep)
    execAction()
    {
        /* let the actor handle it */
        gActor.goToSleep();
    }
;
```

So that if we want to change the way the Sleep action works, we might do better to override the **goToSleep()** method on the player character object (usually **me**). To look up the definition of an existing action, click on the *Actions* link near the left hand end of the top bar of the *Library Reference Manual*. A list of actions will then appear in the bottom left-hand pane, and you can scroll down and click on the one you're interested in.

6.3.2 Actions With Objects

If an action has a direct object, or both a direct object and an indirect object, then we define the action handling on one or both of those objects, generally by using the **dobjFor()** macro on the direct object and the **iobjFor()** macro on the indirect object (just *how* we use them is something we'll come to shortly). But what exactly do we mean by defining the action handling on these objects?

Where an action involves a direct object, or a direct object and an indirect object, the significant stages in handling the action are performed by calling a number of methods on these objects; these are the methods we define with the **dobjFor()** and **iobjFor()** macros. The direct and indirect objects of an action (where they exist) will always be objects derived from the **Thing** class (either of class **Thing** themselves or inheriting from **Thing**). The basic handling for each action therefore needs to be defined on the **Thing** class, even if it's simply a refusal to carry out the action (e.g. displaying a message saying "You can't eat that" in response to the Eat action). It may then be necessary to override this basic action handling on subclasses that need to behave differently, for example to allow objects of class **Food** to be eaten, or stopping **NonPortable** objects from being taken and moved around. Finally, we may often want to override the action handling on individual objects to make them behave in a particular way; for example, if only one green button makes the airlock door slide open, then we need to write special action handling for pressing that particular green button.

We can customize action handling at any of these levels (or introduce a new level of our own by defining custom classes). If we want to change the library's default handling of certain actions, we can modify the action handling on **Thing** or on one of its relevant subclasses; if we need specialized handling just on a particular object, we override the action handling just for that object.

The `dobjFor()` and `iobjFor()` macros can also be used with two pseudo-actions, `Default` and `All`. If we define `dobjFor(Default)` or `iobjFor(Default)` handlers on a class or action, these action handlers will be used for all actions for which we do not provide a more specific action handler. If we define `dobjFor(All)` or `iobjFor(All)` these will be invoked in all cases (although action-specific handling could then be invoked as well if it has not been prevented by the `All` handling).

6.3.3 Stages of an Action

There's no need to override every stage of action handling, but if we were to, our action handling would, in outline, take the following form:

```
banana: Food 'banana*bananas fruit food' 'banana'
     dobjFor(Eat)
     {
          remap() { ... }
          preCond = [ ... ]
          verify() { ... }
          check() { ... }
          action() { ... }
     }
;
```

Where the ... represents the particular code we'd need to write to customize the action handling at each stage.

At a first approximation, the action handling goes through each of the remap, verify, check, and action stages in turn. In fact, any of these stages could stop the action, so that, for example, if we wrote a `verify()` routine that always stopped the action, there would be no need to go on to write `check()` and `action()` routines (they'd never be executed). This is only a first approximation, because the `preCond` property contains a number of objects (called preconditions) defining methods that are called either at the verify stage, or between the verify and check stages.

We'll now look at each stage in turn.

6.3.4 Remap

The purpose of the remap stage is to divert one action into a different action. For example, if we define a desk with a drawer, we might want **open desk** to be treated as **open drawer**. We can achieve that like this:

```
desk: Surface, Heavy 'desk*desks' 'desk'
     "The desk has a single drawer. "
     dobjFor(Open) remapTo(Open, drawer)
;

+ drawer: OpenableContainer, Component 'drawer*drawers' 'drawer'
```

```
;
```

Note how the `remapTo()` macro is used here. If we define an unconditional remapping in this way, there's no point in going on to define any other parts of the same action; **open desk** will be remapped to **open drawer** before any other part of the **open desk** action can be executed. It's also possible to carry out a conditional remapping with the `maybeRemapTo()` macro. This takes an additional first argument, an expression that controls whether the remapping takes place or not. If the expression evaluates to `nil` or `0` the remapping does not occur, otherwise it does. For example, supposing that we think that pulling the drawer should open it, but only if the drawer is not already open. We might achieve this with:

```
+ drawer: OpenableContainer, Component 'drawer*drawers' 'drawer'
   dobjFor(Pull) maybeRemapTo(!isOpen, Open, self)
;
```

Note how in this case we use the `self` keyword to refer to the drawer from a method defined on the drawer.

We can also remap actions that take two objects (i.e. a TIAction), but some special rules apply:

- one TIAction can only be remapped to another TIAction
- one of the objects of the remapped action must be specified as a particular object, while the other must be specified with the placeholder `DirectObject` or `IndirectObject`, meaning the direct object or indirect object of the original action.

For example, suppose we wanted **put something in desk** to remap to **put something in drawer**; we'd do it like this:

```
desk: Surface, Heavy 'desk*desks' 'desk'
     "The desk has a single drawer. "
     dobjFor(Open) remapTo(Open, drawer)
     iobjFor(PutIn) remapTo(PutIn, DirectObject, drawer)
;
```

Note that there's an alternative to using remap which we can use in the particular case when we want one action to behave like another on the same object, namely `asDobjFor()` or `asIobjFor()`. For example if we wanting attacking the desk to behave just like breaking the desk, we could add to the desk definition:

```
      dobjFor(Attack) asDobjFor(Break)
```

Note that although this would have much the same effect in practice as defining `dobjFor(Attack) remapTo(Break, self)`, the underlying mechanism is a little different. Using `remapTo()` cancels the current action and creates a new one; using `asDobjFor()` or `asIobjFor()` continues the current action but makes it use the other action's action-handling routines.

6.3.5 Verify

The verify stage has two purposes:

- To help the parser decide which objects are the most suitable targets for the current command.
- To explain why the command may not be carried out with this object if the verify routine chooses to disallow it.

In the language of TADS 3, the purpose of a verify routine is to decide whether or not an action with this object is *logical*, and how logical or illogical it is. In cases of ambiguity (e.g. **take ball** when there's a red ball, a blue ball, and a green ball all in scope), the parser will choose the most logical object in scope. If it finds a tie for first place (i.e. more than one object has the most logical – or least illogical – score) it will prompt the player to stipulate which object he or she means. In this case 'logical' means 'logical from the perspective of the player' (the point is to try to guess what the player most probably meant). So, for example, if there's a large stone ornamental ball on a plinth, a small red rubber ball lying on the ground, and a golf ball in the player's hand, **take ball** is most likely to refer to the small red rubber ball (the large stone ball is rather obviously untakeable and the player already has the golf ball).

The simplest form of a verify() routine is one that does nothing; far from being pointless this means that the action is allowed to go ahead with this object; it's a perfectly logical choice of object for this command. It's useful to define empty verify methods to allow actions the library would otherwise have ruled out as illogical, for example:

```
banana: Thing 'banana*bananas fruit food' 'banana'
    dobjFor(Eat)
    {
        verify() {}
    }
;

knife: Thing 'knife*knives' 'knife'
    iobjFor(CutWith)
    {
        verify() {}
    }
;
```

We can't eat ordinary things, but we can eat a banana (of course it would have been simpler to define the banana as a `Food` here, but we're just illustrating the principle); in general the library won't let us use things to cut other things with, but a knife presumably can be used for cutting.

It's almost as simple to use verify to rule out an action. In the simplest case we use the `illogical()` macro, which also needs to state why the action is being disallowed;

for example:

```
knife: Thing 'knife*knives' 'knife'
    dobjFor(Eat)
    {
        verify() { illogical('You lack training to swallow a knife safely' ); }
    }
;
```

We can also disallow actions conditionally, depending on the game state, for example:

```
banana: Food 'banana*bananas fruit food' 'banana'
    hasBeenPeeled = nil
    dobjFor(Eat)
    {
        verify()
        {
           if(!hasBeenPeeled)
             illogicalNow('You\'ll have to peel it first. ');
        }
    }
;
```

Note that in this instance we use `illogicalNow()` rather than just `illogical()`; eating a banana isn't illogical *per se*, it's just illogical to attempt it until the banana has been peeled. The point of using a different macro here is that eating the unpeeled banana would be less illogical than attempting to eat the ornamental stone banana on the sculpture, say, so that the we still want the parser to prefer the fruit to the sculpture even when the fruit is unpeeled.

A third kind of verify routine allows an action to proceed, but adjusts its logical ranking, either up or down from the default of 100. For example, if at some point in the game the protagonist is romantically attracted to a particular NPC (let's call her Mary), then other things being equal, she's the most likely target of a Kiss action, so we might define:

```
mary: Person 'mary/woman*women' 'Mary'
     isHer = true
     isProperName = true
     dobjFor(Kiss)
     {
          verify() {  logicalRank(120, 'beloved'); }
     }
;
```

With this definition, **kiss woman** will be taken to mean **kiss mary** even if other women are present, although an explicit **kiss anne** command (say) will still be allowed (provided Anne is present to be kissed). As is apparent from this example, the `logicalRank()` macro takes two arguments: the first is the logical rank score, with 100 being the default, so that giving something a logical rank of more than 100 makes it a likely target of the command, while decreasing it below 100 makes it a possible but not likely target; the library assumes that logical ranks will generally be in the range 50-150. The second argument (in this example 'beloved') is a key; this is an arbitrary string which you can use to define the quality you're ranking, so that if the parser is trying to break a tie it can compare logical ranks assigned to the same key. It seldom matters much in practice in our own game code what we put here, so long as we put something, but it's probably a good idea to get into the habit of using something meaningful.

There are a number of macros we can use within verify routines. As a quick rule-of-thumb, those whose name starts with the letter i disallow the action (with this object) altogether, while the rest allow the action to go ahead with this object but vary the likelihood of the parser choosing it as a target for the command. The complete list (slightly simplified) is:

- `logical` – equivalent to assigning a logical rank of 100, or defining an empty verify statement. This is provided so that we can make it explicit that we're allowing an action, which may be particularly useful when our verify routine contains a number of conditional branches.
- `illogical(msg)` – disallow this action with this object, because the object is never suitable (e.g. trying to cut something with a banana); the *msg* parameter explains why we're disallowing the action (msg can be a single-quoted string or a message property; we'll talk about message properties in a later chapter).
- `illogicalAlready(msg)` – disallow this action with this object because we're trying to bring about a state that already exists (e.g. opening a door that's already open).
- `illogicalNow(msg)` – disallow the action because it's inappropriate while the object is in its present state (e.g. trying to eat an unpeeled banana), or possibly because it's inappropriate while some other part of the game world is in its present state.
- `illogicalSelf(msg)` – disallow the action where the direct and indirect objects are the same and the object can't carry out the action on itself, e.g. **cut knife with knife**.
- `inaccessible(msg)` – disallow the action because the object isn't accessible (even though it's in scope).
- `dangerous` – allow the action to be carried out if the player explicitly insists on it, but not otherwise (e.g. as an implicit action or as the result of the parser choosing a default object). This is intended for actions that the player would perceive as obviously dangerous, such as breaking a glass jar full of poisonous gas, to prevent them being carried out by accident.
- `nonObvious` – similar to dangerous, but intended to prevent a player solving a puzzle by accident by using an unobvious object to carry out a command even though it may in fact be the correct solution, e.g. **unlock case with toothpick**.

Fuller details are available in the other documentation we'll mention at this end of this section. In the meantime there are a few more points to note about verify routines:

- Verify routines should *never* change the game state, and *never* display any text except via the macros just listed. A verify routine may be run several times during object resolution and command execution.
- It's perfectly okay for a verify routine to produce more than one result; the one that counts will be the *least* logical one currently applicable.
- It's therefore safe to use the **inherited** keyword to use the inherited behaviour of a verify routine and then add further cases of your own.
- A verify routine should thus contain nothing apart from one or more of the macros listed above, the **inherited** keyword, and flow control statements (such as **if).**

6.3.6 Check

The only role of a check routine is to disallow actions (if they need to be disallowed). At first sight this may seem the same as ruling out an action with an **illogical** macro at the verify stage. The difference is that ruling out an action at the check stage doesn't affect the parser's choice of object.

For example, suppose the player character is a woman wearing a dress. Removing the dress is not illogical, insofar as the dress is a perfectly sensible target of a Doff command, but we might nevertheless not want to allow it. We could therefore write:

```
me: Actor
;

+ Wearable 'dress*dresses clothes' 'dress'
    wornBy = me
    dobjFor(Doff)
    {
        check()
        {
             reportFailure('It would be quite unseemly to strip in public. ' );
             exit;
        }
    }
;
```

Note the use of the **exit** macro to stop the action in its tracks at this point, and the **reportFailure()** macro to explain why we're stopping it. It's not strictly necessary to use **reportFailure()** - we could legally use a double-quoted string here (unlike verify) – but using **reportFailure()** is a good habit to get into (it helps the transcript work a little better). Using the combination of **reportFailure()** and **exit** in a check method is so common, that Thing defines a **failCheck()** method to do both in a single command; the previous example could be written:

```
+ Wearable 'dress*dresses clothes' 'dress'
    wornBy = me
    dobjFor(Doff)
    {
        check()
        {
             failCheck('It would be quite unseemly to strip in public. ' );
        }
    }
;
```

Check routines should generally be used for no other purpose than this (or to be overridden to do nothing in order to allow an action to go ahead when inheriting from something that would have prevented it), although it is, of course, perfectly legal to rule out an action conditionally in check. For example, if we wanted our player character to be able to remove her dress in her own bedroom but nowhere else, we could rewrite the previous example as:

```
+ Wearable 'dress*dresses clothes' 'dress'
    wornBy = me
    dobjFor(Doff)
    {
        check()
        {
            if(!me.isIn(myBedroom))
              failCheck('It would be quite unseemly to strip in public. ' );
        }
    }
;
```

Check routines should not display text other than to explain why an action is being forbidden, nor should they change the game state (with one possible exception: it's perfectly okay for a check routine to set a flag the sole purpose of which is to show that the check routine has been run, so that we can later test whether the player attempted a certain action even though it did not succeed).

For more guidance on the difference between check and verify, see the article on 'Verify, Check and When to Use Which' in the *TADS 3 Technical Manual*.

6.3.7 Action

Once action processing has survived the remap, verify, check (and possibilities precondition) stages, we're ready to actually carry out the action. That's what the action stage is for: to make the appropriate changes to the game state and report what's happened. This can be as simple or as complicated as we like. At its simplest an action routine may simply report that nothing very much happened as a result of the action:

```
+ Button 'green button*buttons' 'green button'
   dobjFor(Push)
   {
      action() { "You push the green button but nothing happens. ";  }
   }
;
```

More usually, we’d want some change to result from the action. For example, if the green button controls a sliding door, we’d want pushing it to open the door when closed and close the door when open:

```
+ Button 'green button*buttons' 'green button'
   dobjFor(Push)
   {
      action()
      {
          slidingDoor.makeOpen(!slidingDoor.isOpen);
          "You push the green button and the door slides
           <<slidingDoor.openDesc>>. ";
      }
   }
;
```

One complication occurs when the action involves two objects, e.g. **cut banana with knife**. In such a case we have to decide whether to define the action handling on the direct object or the indirect object. Although it would be possible to implement part of the handling on one and part of it on another, this is likely to lead to confusion unless we’re very sure what we’re doing. In general it’s probably a good idea to define the action handling on the object that makes most difference to the outcome. For example, if there’s only one item in the game capable of cutting things, or if all the cutting objects behave in much the same way, but cutting different things (the butter, the banana, the glass case, and Aunt Beatrice, say) has substantially different results, it’s probably best to define the action handling on the direct object of CutWith commands. If however, it makes a huge difference whether you cut things with the butter knife or the dagger or the Magic Diamond Sword, then if may be better to define the action handling on the indirect object (if both the object cut and the object used to cut it with make a significant difference, then we just have to make an arbitrary choice of one or the other and stick with it).

We can more or less put whatever code we like in an action routine, provided it gets the job done. It’s always worth remembering to use the `inherited` keyword where we want the default handling to take place but just want to customize it slightly, e.g.:

```
vase 'delicate antique glass vase*vases'  'antique vase'
    dobjFor(Drop)
    {
         action()
         {
             inherited;
             "You set the glass vase down <i>very</i> carefully. ";
         }
    }
;
```

Here we want **drop vase** to have its normal effect, we just want it reported differently. Since the library uses the `defaultReport()` macro to produce the standard laconic 'Dropped' message, this message will automatically be suppressed in favour of our more appropriate custom message (we can similarly use the `defaultReport()` macro when defining such reports for our own custom actions). There are a number of other macros we can use to generate reports: `mainReport()`, `reportAfter()`, `reportBefore()` and `extraReport()`, but most of the time we can just use a double-quoted string to say whatever we want to say in an action routine. It's also legal to use `reportFailure()` or `failCheck()` in an action routine if we want the outcome to be considered a failure.

Two other macros it's useful to know about in relation to action routines are `nestedAction()` and `replaceAction()`. Both of these can be used to carry out some other action; the difference is that the action routine will continue after `nestedAction()` but not after `replaceAction()` (which stops the current action). For example, suppose we have a button controlling a sliding door, but after the player has discovered that the door can be opened by pressing the button, we want **open door** to be redirected to **push button** provided the door isn't already open. We might write something like this:

```
+ slidingDoor 'sliding door*doors' 'sliding door'
   hasBeenOpened = nil
   makeOpen(stat)
   {
      inherited(stat);
      if(stat)
         hasBeenOpened = true;
   }

   dobjFor(Open)
   {
     verify()
     {
        if(isOpen)
           illogicalAlready('It\'s already open. );
     }
     check()
     {
        if(!hasBeenOpened)
           failCheck('You\'ll have to work out how to open it. ');
     }
     action()
     {
         replaceAction(Push, greenButton);
     }
   }
;
```

6.3.8 Precondition

The final part of action-handling we need to look at is preCond, short for precondition. Often in Interactive Fiction one (relatively mundane) action needs to be carried out in order to allow another one to go ahead. In order to put the banana in the box, I first need to be holding the banana; in order to go through the door, I first need to open it; in order to open the door, I first need to unlock it. If I can't hold the banana, or open the door, or unlock the door, the main action cannot go ahead. In other cases some condition just needs to be true for the main action to proceed; I need to be able to see the book or the rug before reading the book or examining the rug.

These standard conditions are implemented in TADS 3 via **`PreCondition`** objects. These objects instantiate often-used preconditions by defining two methods: **`verifyPreCondition()`** and **`checkPreCondition()`**. The first of these is called at the verify stage, and basically adds further conditions that can rule out an action altogether (e.g. if it's too dark to see by) or can change its logical ranking. The second is called between verify and check, and can carry out an *implicit action* (like taking the banana to allow the player to put it in the box) which allows the main action (putting the banana in the box) to go ahead. If the necessary condition already obtains (the player is already holding the banana), then **`checkPreCondition()`** method has nothing to do. If the necessary condition doesn't hold (the player isn't yet holding the banana, say), the method tries to bring it about through an implicit action (the kind of action that's reported as "(first taking the banana)") and then tests to see if the condition now obtains (since something may have prevented the actor from taking the banana). If it does not, the precondition fails the action; otherwise, it can go ahead.

The library defines a number of precondition objects, the most commonly-used of which include:

- **`objVisible`** **–** the object must be visible to the actor for the action to proceed.
- **`objHeld`** – the actor must be holding the object for the action to proceed (an implicit take action is attempted if not).
- **`objOpen`** – the object must be open for the action to proceed (an implicit open action is attempted if not).
- **`objClosed`** – the object must be closed for the action to proceed (an implicit close action is attempted if not).
- **`objUnlocked`** – the object much be unlocked for the action to proceed (an implicit unlock action is attempted if not).
- **`touchObj`** – the actor must be able to touch the object for the action to proceed.
- **`actorStanding`** - the actor must be standing for the action to proceed (an implicit stand action is attempted if not).
- **`doorOpen`** – like the **`objOpen`** precondition but caters for the case where the actor can't see the door.

For a complete list, see the article we'll refer to shortly.

To use these preconditions, we simply need to list them in the appropriate **`preCond`** property. For example, in order to eat the banana we'd probably need to be holding it, so we might define:

```
banana: Edible 'banana*bananas fruit food'  'banana'
    dobjFor(Eat)
    {
       preCond = [objHeld]
       action()
       {
           "Well, that tasted good! But it's all gone now! ";
           moveInto(nil);
       }
    }
;
```

This has been a very rapid outline of customizing actions, both to avoid the essentials becoming lost in a mass of detail, and also because most of the detail is amply documented elsewhere. We shall examine some of the other details in a later chapter, but in the meantime, if you have not already done so, now might be a good time to read the 'Action Results' article in the *TADS 3 Technical Manual*. If you want more information after that, you could also read the 'Controlling the Action' section of Chapter 4 of *Getting Started in TADS 3*.

6.4 Coding Excursus 10 – Switching and Looping

Now that we've been introduced to action handling, we'll have much more occasion to write procedural code. This thus seems a good point at which to introduce some of the other main coding constructs.

6.4.1 The Switch Statement

We can, if we like, nest if statement to any depth, but when we're basically just testing the same variable against a number of different possible values, it can become a little cumbersome. For example:

m**odify JumpAction**

```
   execAction()
   {
        if(gActor.getOutermostRoom == bedroom)
            "You'd better not, you might wake your Aunt Maude next door. ";
        else if(gActor.getOutermostRoom is in (cellar, lowPassage))
            "Ouch! You bang your head on the ceiling. ";
        else if(gActor.getOutermostRoom == attic)
        {
            "You land back on the rotten floor and fall through to the
             bedroom below; luckily, landing on the bed breaks your fall. ";
             gActor.moveIntoForTravel(spareBed);
             gActor.makePosture(lying);
             gActor.lookAround(true);
        }
        else
            "You jump up and down, uselessly expending energy. ";
   }
;
```

In this kind of case we'd be better off using a **switch** statement. This tests the value of a variable, and then executes a different branch depending which **case** statement is matched. Note that we need to use a **break** statement between one case and the next to prevent falling through, unless we actually want to fall through to the next case (as we do with the cellar). The general form of a switch statement is:

```
switch(expr)
{
    case a: ...
    case b: ...
    default: ...
}
```

There can be as many **case** statement as we like, but only one **default** statement (which defines what happens if no **case** statement is matched). The values following the keyword **case** must be constants. Using a switch statement our example becomes:

```
modify JumpAction
   execAction()
   {
      switch(gActor.getOutermostRoom)
      {
        case bedroom:
            "You'd better not, you might wake your Aunt Maude next door. ";
            break;
        case cellar:
        case lowPassage:
            "Ouch! You bang your head on the ceiling. ";
            break;
        case attic:
            "You land back on the rotten floor and fall through to the
             bedroom below; luckily, landing on the bed breaks your fall. ";
             gActor.moveIntoForTravel(spareBed);
             gActor.makePosture(lying);
             gActor.lookAround(true);
             break;
        default:
            "You jump up and down, uselessly expending energy. ";
            break;
   }
;
```

For more details, see the section on 'switch' in the 'Procedural Code' article in the *TADS 3 System Manual*.

6.4.2 Loops

TADS 3 defines four kinds of loop, one of which (**foreach**) we'll leave to a later chapter. The other three are **while**, **do...while**, and **for**.

The format of the **while** loop is basically

```
while(cond)
   loopBody
```

Where *loopBody* is a single statement or block of statements that continues to be executed while *cond* is true. For example:

```
local i = 0;
while (i <= 10)
{
    i++;
   "<<i>>\n";
}
```

This would cause the numbers 1 to 10 to be displayed in a vertical column.

The `do ... while` loop is similar, except that the test is made at the end. The format is:

```
do
   loopBody
while(cond);
```

For example:

```
local i = 0;
do
{
    i++;
   "<<i>>\n";
}
while (i <= 10)
```

This would do much the same as the first example. The only difference is that if the first statement in each example set i to some value greater than 10, the `do...while` loop would still execute once (so we'd see one number displayed) whereas the `while` loop would not (so we wouldn't see anything displayed from it).

The final loop type is the `for` loop. This is the most complex and powerful of the three. Its general form is:

```
for( initializer; condition; updater)
   loopBody
```

Once again, *loopBody* is a statement or block of statements that is repeatedly executed while the loop is active. The *initializer* initializes the value of one or more loop variables. The loop continues executing while *condition* remains true. The *updater* is used to change the value of the loop variable(s). So, for example, our previous examples could have been written:

for(local i = 0; i <= 10; i++)

```
    "<<i>>\n";
```

With all three types of loop it's essential to make sure that they end somehow, so that we don't end up putting the game in an infinite loop. For example, the following loop would go on forever, causing our game to hang:

```
local i = 0;
while (i <= 10)
{
   "<<i>>\n";
}
```

There is, however, another way out of a loop, and that's to use a **break** statement. The following example will only print the numbers from 1 to 10:

```
local i = 0;
while (i <= 1000)
{
   i++;
   "<<i>>\n";
   if(i >= 10)
      break;
}
```

The **break** statement (usable with all four kinds of loop) takes program execution straight out of a loop. Its complement is the **continue** statement that makes execution jump to the next iteration, skipping over the rest of the loop. The following example will also only print the numbers 1 to 10, although it might cause a bit of a pause after displaying the number 10:

```
local i = 0;
while (i <= 1000)
{
   i++;
   if(i >= 10)
      continue;
   "<<i>>\n";
}
```

For more details of these loops, see the 'Procedural Code' chapter of the *TADS 3 System Manual*.

6.5 Defining New Actions

Being able to modify the responses to existing actions is useful, but most works of IF normally require at least a few completely new actions as well. There are generally three steps to defining an action: (1) defining the new action class; (2) defining the grammar that triggers the action; and (3) writing code to handle the action.

Defining a new action class is generally just a matter of using the appropriate **DefineXXXAction** macro. The name of the macro we need to use is generally the name of the action class preceded by 'Define'. For example, suppose we want to define a new TAction (an action taking a single, direct, object) which will respond to commands like **cross so-and-so** (as in **cross the road** or **cross the bridge**). To define the new action class we'd just write:

```
DefineTAction(Cross);
```

Similarly, if we wanted to define a new `TIAction` (an action taking two objects, a direct object and an indirect object) we'd just define, say:

```
DefineTIAction(OpenWith);
```

If we want to define a new action that takes no objects at all, such as an IAction, we have to do a little more work; or rather we need to combine the first and third steps in the same definition, for example:

```
DefineIAction(Think)
    execAction()
    {
        "You think as hard as you can, but it doesn't seem to do much good. ";
    }
;
```

In practice we might want to code a more interesting and varied response, but the principle remains the same.

The second step is to define the grammar that will match these actions, in other words the pattern of words the player needs to type to make our new action happen. To do that we use a `VerbRule()` macro. For example, to make **cross so-and-so** match our new CrossAction we could define:

```
VerbRule(Cross)
    'cross' singleDobj
   : CrossAction
   verbPhrase = 'cross/crossing (what)'
;
```

Here `singleDobj` is a grammar token (or rather, a macro expanding into one) that matches a single noun, the direct object. If we wanted it to be possible to cross several things at once, we could use `dobjList` here instead, but crossing is the kind of action you can only do to one object at a time, so `singleDobj` seems the better choice here. The first part of the definition thus states that this grammar will match commands consisting of the word **cross** followed by the name of a single noun. The next line, a colon followed by the the action class name, defines the action with which this `VerbRule` is associated. Although we called it `VerbRule(Cross)`, this doesn't automatically associate it with the `DefineTAction(Cross)` we used earlier. The tag we attach to a `VerbRule` is just an arbitrary name (which needs to be unique among `VerbRule` tag names); it is, however, convenient to give it a name identical or at least similar to the corresponding action so we can easily see which **VerbRule** goes with which action.

If a colon followed by an identifier looks a bit like the part of an object definition where we list the classes an object inherits from; that's no accident, we are in fact defining the **VerbRule** (or rather the underlying grammarProd, but we won't let that worry us too much) as being of the **CrossAction** class. This is what associates it with the CrossAction (when we use **DefineTAction(Cross)** to define a new action we in fact define a new action class called CrossAction).

Following the class name, we can define properties and methods in the normal way, but the only property we generally need to define here is the **verbPhrase**; the library uses this to construct message relating to the action such as "What do you want to cross?" or "(first crossing the river)". The format of the **verbPhrase** string is generally 'infinitive/participle (placeholder)'. The infinitive (actually the infinitive less 'to') is the form of the verb that follows 'to' in phrases such as "What do you want to..."; the participle is the form of the verb ending in "ing", and the placeholder is usually the interrogative pronoun ('whom' or 'what') we want used in posing questions about the action ("Whom do you want to ask?" or "What do you want to cross?").

We might want to tweak this **VerbRule** a little further, since there are more ways of phrasing the command than just **cross street**; we might, for example, want the phrasing **walk across street** and **go across street** to trigger the same action. We can do this by using a vertical bar (|) to separate alternatives, and parentheses to group them, so that our **VerbRule** would become:

```
VerbRule(Cross)
    ((('walk' | 'go') 'across') | 'cross') singleDobj
   : CrossAction
   verbPhrase = 'cross/crossing (what)'
;
```

If we're defining an **IAction** our **VerbRule** can generally be a bit simpler:

```
VerbRule(Think)
   'think' | 'ponder' | 'cogitate'
   : ThinkAction
   verbPhrase = 'think/thinking'
;
```

Conversely, if we're defining a TIAction we need a token for the indirect object as well as the direct object, for example:

```
VerbRule(OpenWith)
   'open' dobjList 'with' singleIobj
   : OpenWithAction
   verbPhrase = 'open/opening (what) (with what)'
;
```

Here the use of **dobjList** allows us to try to open several objects at once with the same indirect object. It would also be legal (though in practice far less usual) to use `iobjList`, but we cannot use both `dobjList` and `iobjList` in the same `VerbRule.` A command like **open the soup can, the beer bottle, and the paint tin with the can opener, the bottle opener and the screwdriver** would just be too convoluted to handle.

The third stage is to define what the action does. If the action doesn't have any objects, we'll have done that already in the `execAction()` method of the action class when we defined it (see above). If it does have any actions we must define at least minimal handling on the Thing class (to trap attempts to try the action out on objects we never intended it for). For example:

```
modify Thing
    dobjFor(Cross)
    {
         preCond = [touchObj]
         verify() { illogical(cannotCrossMsg); }
    }
    cannotCrossMsg = '{That dobj/he} {is} not something {you/he} can cross. '

    dobjFor(OpenWith)
    {
         preCond = [touchobj]
         verify() { illogical(&cannotOpenMsg) }
    }

    iobjFor(OpenWith)
    {
         precond = [objHeld]
         verify() { illogical(cannotOpenWithMsg); }
    }
    cannotOpenWithMsg = '{You/he} cannot open anything with {that iobj/him}. '
;
```

There are several things to note about this example. First, we *could* have just defined the failure messages directly, for example with:

```
    dobjFor(Cross)
    {
         preCond = [touchObj]
         verify()
         {
            illogical('{That dobj/he} {is} not something {you/he} can cross. ');
         }
    }
```

The reason for not doing it that way is that it makes it so much easier to customize the message for special cases, for example:

```
river: Fixture 'river*rivers' 'river'
    cannotCrossMsg = 'You can\'t walk on water! '
;
```

This is rather more convenient than having to redefine the `dobjFor(Cross) verify()`

method on the river object.

Note, however, that when we came to define **dobjFor(Open)** we preceded **cannotOpenMsg** with an ampersand: **illogical(&cannotOpenMsg)**. We'll give a full explanation of this is a later chapter; the brief explanation is that we're borrowing the library's **cannotOpenMsg**, which is defined on **playerActionMessages**, and this is how we do it (if we use a property pointer in this context, the library assumes we want to use a property of the appropriate message object).

Another point to note is that our messages contain lots of strange looking pieces of text in curly braces, like **{you/he}** or **{That dobj/he}**. These are *message parameter strings*. When the text is actually displayed the library substitutes text appropriate to the circumstance. For example **{you/he}** becomes just 'you' if the player character is carrying out the action, or the name of the actor, e.g. 'Bob', if an NPC is carrying out the action. Similarly **{That dobj/he}** becomes either 'That' or 'Those' depending on whether the direct object is singular or plural. The string **{is}** expands into either 'is' or 'are' in order to agree with its subject, and we can use **{s}** or **{es}** at the end of other verbs to secure similar agreement, e.g. '**{You/he} put{s} down {the dobj/him}**. ' Using these message parameter strings helps makes our responses as general as possible. Other commonly useful ones include:

- **{the dobj/he}** – the name of the direct object preceded by the definite article ('the')
- **{a dobj/he}** – the name of the direct object preceded by the indefinite article ('a' or 'an')
- **{it dobj/he}** – the correct pronoun for the direct object in the subjective case ('he', 'she', 'it' or 'they')
- **{it dobj/him}** – the correct pronoun for the direct object in the objective case ('him', 'her', 'it' or 'them')

These all work with iobj or actor in place of dobj (to refer to the indirect object or the actor), and can be made to work with any object whatsoever provided it has an appropriate parameter name. For example, if we define:

```
+ banana: Food 'banana*bananas fruit food' 'banana'
    globalParamName = 'banana'
;
```

We can then use parameters substitution strings like **{the banana/he}** or **{it banana/him}**. We can also temporarily assign a parameter string to a local variable representing an object using the **gMessageParams()** macro, for example:

```
    talkAbout(obj)
    {
        gMessageParams(obj)
        "{The obj/he} {is}, in your opinion, utterly hideous. ";
    }
```

Note that if we start a message parameter string with a capital letter, its substitution will also start with a capital letter.

For the full story on message parameters, including a list of all the ones the library defines, see the article 'Message Parameter Substitutions' in the *TADS 3 Technical Manual*. If you're thinking of writing a game in the past tense, or one that switches between tenses, you might also like to read the article on 'Writing a Game in the Past Tense' also in the *Technical Manual*, but unless that's an urgent concern for you, you might want to leave it for now.

The final thing to note about our example (now a couple of pages back) is that we assumed that you have to be able to touch something in order to cross it or open it with something else, but you have to be holding something in order to use it to open something else with.

One further stage would be to define the handling on objects or classes where we want our new actions to actually do something. For example, we might define a Crossable class for which the command **cross x** takes us to some other location (e.g. the other side of the bridge):

```
class Crossable: Enterable
    dobjFor(Cross)
    {
        verify() { }
        action()
        {
             "{You/he} set{s} out across {the dobj/him}. ";
              replaceAction(TravelVia, connector);
        }
     }
;
```

Of course, if there was only one crossable object in the entire game, we probably wouldn't bother to do this; we'd simply define the handling directly on that object; but as soon as we want similar handling on more than one object it's worth considering defining a new class (or modifying an existing one).

We have here given a somewhat compressed account of defining new actions; for a fuller account, read the article on 'How to Create Verbs' in *The TADS 3 Technical Manual*.

Exercise 13: Now that you've seen how to implement actions, you can finish off the previous exercise. Return to your kitchen and make the can opener able to open the can of soup. Put some soup in the can that can be poured into appropriate objects (but not elsewhere). Implement a pencil sharpener and some pencils, so that only pencils can be put in the sharpener, and the sharpener actually sharpens the pencils. Define some grammar so that it's possible to hang an apron on the peg. Customize eating the cake. If any other ideas occur to you, by all means try them too! Then compare your results with the containers.t example.

7 Knowledge

7.1 Seen and Known

7.1.1 Tracking What Has Been Seen

It is sometimes useful to keep track of what objects the player character has seen, and which he or she knows about. This can be particularly useful when we come to implement conversations (where what the player knows may well affect what's said) or hint systems, but it can be relevant to other aspects of the game besides.

A TADS 3 game keeps track of what the player character has seen. By default the **seen** property of every object is set to true when the player sees it. The library is pretty good at catching most situations in which a player first sees something, but it may miss one or two, in particular when we move an object into the player's location with **moveInto()**. The obvious way to mark an object as having been seen in such a situation (assuming the player character can see it) is simply to set the **seen** property to true. A safer way is to use **gPlayerChar.setHasSeen(obj)** (where *obj* is the object the player character has just seen); this can be abbreviated to the macro **gSetSeen(obj)**. Likewise, while we might test **obj.seen** to see whether *obj* has been seen, it's probably a good idea to get into the habit of using **gPlayerChar.hasSeen(obj)** or **me.hasSeen(obj)**.

The reason for this is that while the **seen** property is the default property used by the library to track what the player character has seen, we can change it to something else. The reason we might want to do this is to track what NPCs have seen separately from what the player character has seen; by default the library uses the **seen** property for every actor in the game, although it only actively tracks what the player character has seen. By default, then, **actor.hasSeen(obj)** will return the same value for every actor in the game, which will be the correct value only for the player character. Likewise, **actor.setHasSeen(obj)** will set the same property (**seen**) for every actor in the game, which is almost certainly not what we want if we're bothering to track what actors other than the player character have seen.

If we want to track what different actors have seen (which, in the majority of games, we probably won't) we can redefine what property to use for the purpose. We do this by changing the actors' **seenProp** property to something other than **&seen**, the default. For example, if want to keep track of what two NPCs, Bob and Carol, have seen, we might define:

```
bob: Person 'bob/man'  'Bob'
    isProperName = true
    isHim = true
    seenProp = &bobHasSeen
;
```

```
carol: Person 'carol/woman'  'Carol'
    isProperName = true
    isHer = true
    seenProp = &carolHasSeen
;

modify Thing
   bobHasSeen = nil
   carolHasSeen = nil
;
```

Note the ampersands (&) before the property names here. If we wrote `seenProp = bobHasSeen` we'd be setting the value of `bob.seenProp` to the value of `bob.bobHasSeen`, which isn't what we want at all. What we want to do is to tell TADS 3 to use the `bobHasSeen` property of `Thing` to keep track of what things Bob has seen; for this purpose we need to use a property *pointer*, which we obtain by preceding the property name with &.

Once we've done this `bob.setHasSeen(obj), carol.hasSeen(obj)` and the like will use the appropriate properties of Thing, so we can keep track of what Bob and Carol have seen separately from what the player character has seen. Note, however, that the library won't actually mark things as seen by Bob and Carol for us; that's something we'll have to take care of for ourselves by calling `bob.setHasSeen(obj)` and `carol.setHasSeen(obj)` whenever Bob and Carol see things.

7.1.2 Tracking What Is Known

People don't necessarily have to have seen something to know about it, so we can keep track of what the player character (and optionally, other NPCs) know about separately from what they have seen. `Thing` defines a `known` property, analogous to the `seen` property which we have just met. We can set the `known` property for the player character using `gPlayerChar.setKnowsAbout(obj)` or the macro `gSetKnown(obj)`. We can similarly test what the player character knows about using `gPlayerChar.knowsAbout(obj)` or, often enough, `me.knowsAbout(obj)`. By default, `known` is `nil` on everything, but if the player character starts out the game knowing about several things, we can define `known` as `true` on those objects.

Although we can know about things without having seen them, once we've seen them, we know about them (at least, to the extent of knowing that they exist, which is the kind of knowledge TADS 3 effectively models). Thus, `gPlayerChar.knowsAbout(obj)` returns true *either* if `known` is true, *or* if the player character has seen obj, *or* if the player character can currently see obj.

Just as we can keep separate track of what different NPCs have seen, we can also keep track of what they know, this time by overriding their `knownProp`:

bob: Person 'bob/man' 'Bob'

```
    isProperName = true
    isHim = true
    seenProp = &bobHasSeen
    knownProp = &bobKnows
;

carol: Person 'carol/woman'  'Carol'
    isProperName = true
    isHer = true
    seenProp = &carolHasSeen
    knownProp = &carolKnows
;

modify Thing
   bobHasSeen = nil
   bobKnows = nil
   carolHasSeen = nil
   carolKnows = nil
;
```

Note that even if we're not particularly interested in tracking what NPCs have seen, we have to define a separate `seenProp` for them if we want to track their knowledge separately (or else test the `bobKnows` and `carolKnows` properties directly). The reason for this is that `actor.knowsAbout(obj)` will be true, as we've just said, either if the appropriate `knownProp` is true or if the appropriate `seenProp` is true. But if we hadn't overridden `bob.seenProp`, then it would still be `seen`, which keeps track of what the player character has seen; this would mean that `bob.knowsAbout(obj)` would be true for every *obj* that the player character has seen.

One way round this if we want to keep track of what NPCs know about, but not what they have seen (which may be quite a common requirement), is to override `seenProp` for the player character only, e.g.:

```
me: Actor
    seenProp = &meHasSeen
;

modify Thing
    meHasSeen = nil
;
```

If we do that, the player character will use a different property to track what s/he has seen from that used by all NPCs (which will still be using `seen`). There is then no need to define a separate `seenProp` for the NPCs unless we actually want to track what they've each individually seen. But then we must remember to use `gSetSeen(obj)` and `me.hasSeen(obj)` to set and test what the player character has seen, rather than using the `seen` property. This is one reason why it's good to get into the habit of using these methods rather than manipulating the `seen` property directly. Another is that if we're half-way through a game and then decide we want to start tracking NPC knowledge separately it will be so much easier if we haven't used `known` and `seen` directly in our code up to that point.

7.1.3 Revealing

There is one more mechanism for keeping track of what is known in a TADS 3 game, which we may call *revealing.* This simply lets us declare an arbitrary string tag as having been revealed, and later test whether or not it has been revealed. To declare something as revealed we simply use the **`gReveal()`** macro, in the form **`gReveal('tag')`**, where 'tag' can be any string we like. To test whether something has been revealed we use the **`gRevealed()`** macro, in the form **`gRevealed('tag')`**. We can also reveal something when a string is displayed using **`<.reveal tag>`**. For example:

```
"<q>Have you heard about the lighthouse?</q> Bob asks anxiously.
    <.reveal lighthouse>";

...

if(gRevealed('lighthouse'))
  "<q>Tell me about the lighthouse,</q> you ask. ";

...

box: OpenableContainer 'box*boxes' 'box'
  dobjFor(Open)
  {
     check()
     {
        if(isStuck)
          failCheck('Something seems to be stopping it open.
            <.reveal box-stuck>');
     }
     action()
     {
        inherited;
        gReveal('box-opened');
     }
  }
  isStuck = true
;
```

As these examples suggest, this mechanism is probably most useful for conversation (for which it was devised) and hints (the hints system, which we'll look at in a later chapter, could display a hint about getting the box open once 'box-stuck' had been revealed and remove the hint again once 'box-opened' had been revealed). But of course we're entirely free to use this mechanism for any purpose we find useful.

7.2 Coding Excursus 11 – Comments, Literals and Datatypes

This is a convenient point at which to tie up a few loose ends, covering a number of things that have been presupposed up to now without being formally explained, and introducing one or two new things it's useful to know about when writing TADS 3 code.

7.2.1 Comments

We can insert a comment into TADS source code in one of two ways. A single line comment is any text starting with `//` and running on to the end of the line. A block comment is any text starting with `/*` and ending with `*/`. Block comments may not be nested. Comments are ignored by the compiler, and so can contain anything we like. We can (and probably should) use comments both to explain our code to others (if anyone else might read it) and, perhaps even more importantly, to explain it to ourselves, or at least to remind ourselves what we were trying to do, why and how.

```
// This is a single-line comment.

local var = nil; // this is another single-line comment.

/* This is a block comment spanning a single line */

/* This is a block comment spanning several lines;
    it can go on for as long as we like, but we can't
    nest another block comment inside it, as the block
   comment will be assumed to come to an end as
   soon as the compiler encounters */
```

If we are using the editor built into Windows Workbench, it can automatically format block comments neatly for us. It can also add and remove `//` comment markers to the beginning of a selected set of lines, this can be useful for 'commenting out' blocks of source code, i.e. temporarily disabling blocks of code for testing or debugging purposes.

7.2.2 Identifiers

An *identifier* is the name of an object, class, function, property, method, or local variable. An identifier must start with an alphabetic character or underscore, which must be followed by zero or more alphabetic characters, underscores, or the digits 0-9. TADS 3 identifiers are case-sensitive, so that Apple, apple, and aPPle would refer to three different things. The normal convention is that class names and macro names start with capital letters, except for macros that behave like pseudo-global variables, which start with a lower case g.

For further information see the articles on 'Naming Conventions' and 'Source File Structure' in the *TADS 3 System Manual*.

7.2.3 Literals and Datatypes

TADS 3 recognizes the following datatypes, represented by the following kinds of literal values:

- **`nil`** and **`true`**: where **`nil`** is a false or empty value.
- Integer: -2147483648 to + 2147483647

- Hexadecimal: 0xFFFF
- Enumerators: e.g. `enum blue, red, green`
- Property ID: `&myProp`
- Function Pointer: e.g. `func`
- List: `[item1, item2, item3, item4, ... itemn]`
- BigNumber: e,g, 12.34 or 1.25e9; can store up to 65,000 decimal digits in a value between 10^{32767} and 10^{-32767}.
- String: an ordered set of Unicode characters. A string constant is written by enclosing a sequence of characters in single quotation marks, e.g. `'Hello World! '`

We've already met strings, integers, nil and true; we'll say more about lists in the next chapter, and something about Enumerator and Property IDs below. BigNumber is one of those things that's nice to have, but which we probably won't use much in Interactive Fiction; for more information see the article on BigNumber in section IV of the *TADS 3 System Manual*. For more information on TADS 3 datatypes in general, see the article on 'Fundamental Datatypes' in the *System Manual*.

7.2.4 Determining the Datatype (and Class) of Something

It's often useful to be able to determine what type of data something is. We can do this with the function `dataType(val)`, where *val* is the data item we want to test. This function returns one of the following values:

• `TypeNil`	nil
• `TypeTrue`	true
• `TypeObject`	object reference
• `TypeProp`	property ID
• `TypeInt`	integer
• `TypeSString`	single-quoted string
• `TypeDString`	double-quoted string
• `TypeList`	list
• `TypeCode`	executable code
• `TypeFuncPtr`	function pointer
• `TypeNativeCode`	native code
• `TypeEnum`	enumerator

If an identifier turns out to be an object, we can also determine its class using the methods `ofKind`() and `getSuperclassList().` The method `obj.ofClass(cls)` returns true if *obj* inherits from *cls* anywhere in its inheritance hierarchy. The method `obj.getSuperclassList()` returns a list of the classes with which *obj* was defined.

For example, suppose we had (in outline) the following definition:

```
box: Lockable, OpenableContainer
;
```

Then `box.getSuperclassList()` would return `[Lockable, OpenableContainer]`, while `box.ofKind(Lockable)`, `box.ofKind(OpenableContainer)`, `box.ofKind(Container)` and `box.ofKind(Thing)` would all return `true` (because `OpenableContainer` descends from `Container` which in turn descends from `Thing`). On the other hand, `box.ofKind(Food)` or `box.ofKind(Person)` would both return `nil`.

Incidentally, there is also a `setSuperclassList()` method which allows us to change the superclass list of an object at run-time. For example, suppose `handle` starts out as a component of `briefcase`, but it can be broken off to form a separate item. We might then want `handle` to perform like an ordinary `Thing`, and we could use `handle.setSuperclassList([Thing])` to bring this about.

One other thing we may wish to do is to is to determine the datatype of a property without evaluating that property. We can do that with the `propType()` method. We call this on the object or class we're interested in, passing a property pointer as the single argument. The return value is one of the `TypeXXXX` values listed above. For example, we could use `box.propType(&name)` to determine whether the `name` property of `box` was simply a single-quoted string, or a piece of code (which might return a single-quoted string).

For further details see the chapters on Reflection, Object and TadsObject in the *Library Reference Manual*.

7.2.5 Property and Function Pointers

If we precede the name of a property or method with an ampersand we turn it into a property pointer. If we give the name of a function without its argument list or any brackets we obtain a function pointer. These pointers are useful when we want a reference to the property or function itself rather than whatever the property, method, or function evaluates to.

When we do want to evaluate (or execute) the property or method, we surround the pointer name in parentheses and then follow it with the argument list.

For example, suppose we define an object with a number of different methods so:

myObj: object

```
    double(x) { return x * 2; }
    triple(x) { return x * 3; }
    quadruple(x) { return x * 4; }
    calculate(prop, x)  {   return self.(prop)(x);   }
;
```

The statement `local a = myObj.calculate(&triple, 2);` will set a to 6. So will the following pair of statements:

```
local meth = &triple;
local a = myObj.(meth)(2);
```

This is used, for example, in the definition of `hasSeen(obj)`, which is defined as:

```
hasSeen(obj) { return obj.(seenProp); }
```

Note the difference between that and

```
hasSeen(obj) { return obj.seenProp; }
```

Which would erroneously return a pointer to the `seen` property (or whichever other property had been defined), instead of the *value* of the `seen` property.

We thus use property pointers when we want to reference properties (or methods) *indirectly*, typically when we need to write code that might use more than one property of some object, but we don't know which property it will be.

A function pointer is similar, but the syntax is a little different. To obtain a function pointer, we don't precede the function name with an ampersand, we just omit the brackets and the argument list following it. Thus with the following definition:

```
halve(x)
{
    return x/2;
}

doSomething()
{
   local a = halve(4);
   local f = halve;
   local b = f(6);
}
```

When `doSomething()` executes, a will evaluate to 2, f will evaluate to a function pointer referencing the `halve()` function, and b will evaluate to 3.

There's one subtlety to note here; we can assign a function pointer to an object property, but it may not work quite as we expect:

```
myObj:
   funcPtr = halve;
   half(x) { return (funcPtr)(x); } // This won't work!
;
```

This will compile, but will probably produce a run-time error if we call **myObj.half()**. To make it work as we want, we first need to store the function pointer in a local variable:

```
myObj:
   funcPtr = halve;
   half(x)
   {
     local f = funcPtr;
     return (f)(x); // but this is fine.
   }
;
```

7.3 Enumerators

It is sometimes useful to have constants with meaningful symbolic names. One way we can do this is by defining a number of macros, e.g.:

```
#define red 1
#define blue 2
#define green 3
```

This is useful if we want our constants to have numerical values, and the numeric values are meaningful, but if we just want to test whether some variable or property is equal to some symbolic constant value, we can use enumerators instead. In this case, we could just define:

```
enum red, blue, green;
```

We can assign these values to properties and variables, and test for equality or inequality, e.g.:

```
local colour = blue;

if(colour == blue)
    "It's blue! ";

if(colour != red)
   "It's not red! ";
```

That’s just about all there is to enumerators, but there are few further points worth noting:

- A **enum** statement is a top-level statement that can appear anywhere outside an object, class or function definition.
- Enumerators are a distinct datatype; enumerators do not have a numerical value, and they cannot be mixed with numbers in arithmetic operations or comparisons.
- There is no relation between enumerators apart from the fact that they *are* all enumerators, and so can legally be compared with one another for equality or

inequality. Declaring several enumerators in one statement does not establish any particular relationship between them.

- Enumerator constants can be used in the case parts of a switch statement provided the switch variable is of enumerator type.

For further information, see the article on ‘Enumerators’ in Part III of the *System Manual*.

7.4 Topics

Earlier in the chapter we saw how we could track the player’s (and optionally other characters’) knowledge of the things in the game. But physical objects aren’t the only things people know about (or can think about, discuss, look up and so forth). People can also know about (or think about, discuss, look up and so forth) abstract topics such as the weather, Chinese politics, the meaning of life, astronomy, and sympathetic magic. If any of these figure in our game, we need to represent them somehow, but they’re not physical objects. For this purpose we use the `Topic` class.

There’s only one property we need to define on a Topic, namely its `vocabWords` (which works in precisely the same way as the `vocabWords` property on Thing). So we might, for example, define:

```
tWeather: Topic vocabWords = 'weather';
tChinesePolitics: Topic vocabWords = 'chinese politics';
tMeaningOfLife: Topic vocabWords = 'meaning/life';
```

Since we always need to define the `vocabWords` property on a `Topic`, as you might imagine we can do so by means of a template:

```
tWeather: Topic 'weather';
tChinesePolitics: Topic 'chinese politics';
tMeaningOfLife: Topic 'meaning/life';
```

There is, by the way, no need to start the name of `Topic` objects with the letter t, but it’s often useful to be able to distinguish `Topics` from physical objects in out code, so it’s a good idea to adopt some such convention (you might prefer to use the slightly more explicit `top` as the identifying prefix, for example).

The other commonly useful property `Topic` defines is `known`, which has the same meaning as the `known` property on `Thing`. Whereas `Thing.known` is nil by default, `Topic.known` starts out as true, so that if there are topics the player character starts the game not knowing about, we need to change `known` to nil on those topics. We can use `gSetKnown()` and all the rest with `Topics` as well as `Things`, but we need to remember that if we override `knownProp` on any actor(s), we need to make the corresponding changes (to allow for the new `knownProp`) on both `Thing` and `Topic`.

There are two other kinds of thing besides abstract topics we can usefully implement as **`Topics`**. The first is physical objects and people who are mentioned in the game but don't actually appear within it as objects in their own right; for example our game may mention William Shakespeare or the planet Uranus or the lost Ark of the Covenant without any of them making any physical appearance in the game; such objects are probably best implemented as **`Topics`**. Topics can also be useful when we want to talk about a group of objects that are implemented in the game; suppose, for example, that our game implements a red ball, a blue ball, a green ball and an orange ball, but at some point we want our player character to be able to discuss coloured balls in general with some NPC; it may well prove convenient to define a **`tColouredBalls Topic`** to do the job.

At this point it's probably worth mentioning that neither the **`singleTopic`** grammar token nor the **`gTopic`** pseudo-variable directly references a **`Topic`** object. They instead refer to a **`ResolvedTopic`** object. Or to put it a bit more carefully, when we define an action that uses the **`singleTopic`** token in its **`VerbRule`** (a **`TopicAction`** or **`TopicTAction`**), the object matching the **`singleTopic`** token, obtainable through the **`gTopic`** pseudo-variable, will be of class **`ResolvedTopic`**.

If we want to get at the actual simulation object or **`Topic`** that was (probably) matched, we can use the **`getBestMatch()`** method, i.e. **`gTopic.getBestMatch()`**. This is only *probably* the simulation object or **`Topic`** in question, since a **`ResolvedTopic`** actually maintains three lists of possible matches (in its **`inScopeList`**, **`likelyList`** and **`otherList`** properties) and **`getBestMatch()`** somewhat arbitrarily returns the first item from the 'best' of these lists that have anything in them. This is generally good enough for most purposes, however. For more details, look up **`ResolvedTopic`** in the *Library Reference Manual*.

To get at the original text the player typed that the **`ResolvedTopic`** is matching, we can use the **`getTopicText()`** method, i.e. **`gTopic.getTopicText()`**. We can use the macro **`gTopicText`** to return this value, converted to lower case.

Finally, note that a **`TopicAction`** or **`TopicTAction`** will always succeed in returning a **`ResolvedTopic`** even if what the player typed matches no **`Thing`** or **`Topic`** defined in the game. In this case **`gTopic.getTopicText()`** will return that part of the player's command that matched the **`singleTopic`** token, but **`gTopic.getBestMatch()`** will be nil.

7.5 Coding Excursus 12 – Dynamically Creating Objects

So far, all the objects we've encountered have been statically defined in our source code. But it's also possible to create objects on the fly at run-time. At its simplest this is just a matter using the keyword **`new`** plus the class name.

For example, suppose we wanted to create an apple tree that goes on dispensing apples for as long as the player attempts to pick them. We could do something like this:

```
DefineTAction(Pick);

VerbRule(Pick)
   'pick' singleDobj
   : PickAction
   verbPhrase = 'pick/picking (what)'
;

modify Thing
  dobjFor(Pick)
  {
     preCond = [touchObj]
     verify() { illogical(cannotPickMsg); }
  }
  cannotPickMsg = '{That dobj/he} {is} not something {you/he} can pick. '
;

class Apple: Food 'apple*apples' 'apple'
  isEquivalent = true
;

orchard: OutdoorRoom 'orchard'
  "An apple tree grows in the middle of the orchard. "
;

+ tree: Fixture 'apple tree*trees' 'apple tree'
;

++ Component 'apple/apples' 'apple'
   dobjFor(Pick)
   {
       verify() { }
       action()
       {
          local apple = new Apple;
          apple.moveInto(gActor);
          "You pick an apple from the tree. ";
       }
   }
;
```

Here the `Component` represents the apples still on the tree. The command **pick apple** will select this object in preference to any apples that have already been picked (since picking a picked apple is illogical); the `actionDobjPick()` method will then create a new `Apple` object and move it into the player's inventory. In principle the player could go on picking apples forever; in practice the game will probably start grinding to a halt after a few dozen apples have been picked.

When we use the `new` keyword to create an object the object's `construct()` method is called immediately after it has been created. This method can take as many parameters as we like, which can be used to initialize the object. For example, suppose we wanted to be able to create a number of different pieces of fruit dynamically, we could define a `Fruit` class thus:

```
class Fruit: Food
    construct(fruitName, nutrValue)
    {
       name = fruitName;
       nutritionValue = nutrValue;
       vocabWords = name + '*fruit ' + 'name' + 's';
       initializeVocabWith(vocabWords);
    }
    nutritionValue = 0

    dobjFor(Eat)
    {
       action()
       {
          "You eat <<theName>>; it tastes jolly good. ";
          gActor.strength += nutritionValue;
          moveInto(nil);
       }
    }
;
```

We could then create different pieces of fruit with code like:

```
  local x = new Fruit('banana', 2);
  local y = new Fruit('apple', 3);
  local z = new Fruit('orange', 4);
```

As an alternative, we could use the `createInstance()` method, called directly on the Fruit class:

```
  local x = Fruit.createInstance('banana', 2);
  local y = Fruit.createInstance('apple', 3);
  local z = Fruit.createInstance('orange', 4);
```

For more information, see the chapters on ‘Dynamic Object Creation’ and ‘TadsObject’ in Parts III and IV of the *System Manual*.

7.6 Consultables

One place where we might look for knowledge, in works of IF as well as in real life, is in books and book-like objects. These are the kinds of thing that can be used in commands like **consult cookery book about pancakes** or **look up tads in encyclopaedia**.

The mechanism TADS 3 provides for implementing such objects uses two classes of object: `Consultable` to represent the book (or other reference work) we’re consulting, and `ConsultTopic` to represent the topics we want the player to be able to look up. We can also define a `DefaultConsultTopic` to provide a catch-all response when the player tries to look up something we haven’t provided for. We then locate the `ConsultTopics` (and the `DefaultConsultTopic`) inside the `Consultable`.

A `ConsultTopic` can be matched either on an object (a `Thing` or `Topic`), or on a regular expression. If we want to it to match on an object, we define its `matchObj` property to be the object in question; if we want it to match on a regular expression we instead define its `matchPattern` property to be a single-quoted string containing the regular expression we want to match. We can, if we like, define both these properties, and then the `ConsultTopic` will match on either (if you're not at all familiar with regular expressions, don't worry about them just yet; if you are, but want to know how they're implemented in TADS 3, look at the 'Regular Expressions' chapter in Part IV of the *System Manual*). The `matchObj` property can also contain a list of objects; the `ConsultTopic` will then match on any one of those objects.

The information that's to be displayed when the player looks up a particular topic is defined in the `ConsultTopic's topicResponse` property. If we want a topic to be only conditionally available, we can set its `isActive` property to the relevant condition (we could, for example, use this to prevent the player from looking up something he's not meant to know about yet).

This should become clearer with a couple of examples:

```
+ Readable, Consultable 'green book*books' 'green book'
   readDesc = "It's rather too long to read from cover to cover, but you
     could try looking up particular topics of interest. "
;

++ ConsultTopic
      matchObj = tWeather
      topicResponse = "The weather in these parts is frequently variable. "
;

++ ConsultTopic
      matchObj =[redBall, greenBall]
      topicResponse = "According to the book, both the green ball and the
         red ball are pretty much round. "

++ ConsultTopic
      matchPattern = '<alpha>{1,3}<digit>{1,3}'
      topicResponse = "According to the green book this could the serial
        number of a type 4 widget-spangler. "
;

++ DefaultConsultTopic
      topicResponse = "The book doesn't seem to have anything to say on that
       topic. "
;
```

If the player issues the command **lookup weather in green book** or **consult green book about the weather** then (assuming the `tWeather Topic` has been suitably defined), the game will respond with "The weather in these parts is frequently variable." If the player looks up the green ball or the red ball in the book, s/he'll get the message about the balls being round. If the player tries looking up **abc123** or some other combination of one to three letters followed by one to three digits s/he'll get the response about the widget-spangler. Trying to consult the green book about anything else will be met with the default response saying that the book doesn't have anything to say on the topic. In a real Consultable we'd probably provide more responses on a more coherent range of topics.

The definition of a large number of ConsultTopics can be made easier (as ever) using a template. We can define the matchObj using @ followed by a single object, or a list of objects in square brackets, or else the matchPattern in single quotes. We can then give the topicResponse simply as a double-quoted string. Using the template, the ConsultTopics defined above can become just:

```
++ ConsultTopic @tWeather
     "The weather in these parts is frequently variable. "
;

++ ConsultTopic [redBall, greenBall]
      "According to the book, both the green ball and the
         red ball are pretty much round. "

++ ConsultTopic '<alpha>{1,3}<digit>{1,3}'
      "According to the green book this could the serial
        number of a type 4 widget-spangler. "
;

++ DefaultConsultTopic
      "The book doesn't seem to have anything to say on that
       topic. "
;
```

Exercise 14: Create your own Consultable object (a book or timetable or anything else you like) with a number of entries. Don't worry about using regular expressions to match ConsultTopics unless you're reasonably comfortable with them. If you need a bit more help look up Consultable and ConsultTopic in the *Library Reference Manual*; you may also find it helpful to look up TopicEntry there as well, along with the TopicEntry template.

8 Events

8.1 Fuses and Daemons

It’s often useful to be able to schedule an event to happen at some point in the future, or to carry out a routine every turn (or every so many turns). For this purpose we can use Fuses and Daemons.

In TADS 3, Fuses and Daemons are created as dynamic objects. We set up a Fuse with a command like:

```
new Fuse(obj, &prop, n);
```

or

```
fuseID = new Fuse(obj, &prop, n);
```

Where `fuseID` (or whatever name we want to use) is typically a property we’re using to store a reference to the Fuse, if we need one. With these definitions the *prop* property of the *obj* object will be executed after *n* turns. If n is 1, `obj.prop` will be executed on the next turn. It n is 0, the fuse will fire on the same turn; this can be useful if we want to set something up to happen at the end of the current turn. For example, we might define:

```
dynamite: Thing 'dynamite/stick' 'stick of dynamite'
     dobjFor(Burn)
     {
         verify() {}
         action()
         {
              new Fuse(self, &explode, 3);
              "You set the dynamite alight. ";
         }
     }
     explode()
     {
          "Bang! ";
          moveInto(nil);
     }
;
```

With this definition the dynamite will explode three turns after it is set alight. If the player should find some way to extinguish it in the meanwhile, we need to find some way to disable the fuse. If we’d stored a reference to the Fuse we could do that most simply with

```
fuseID.removeEvent();
```

If we hadn’t stored a reference to it, we could still disable the Fuse with:

```
eventManager.removeMatchingEvents(dynamite, &explode);
```

In a full implementation of the dynamite, we'd probably do something more dramatic when it exploded than just saying "Bang!" and removing the dynamite from play, but in any case we shouldn't report what happens unless the player character is there to see it. If the player lights the dynamite and then immediately heads off to a remote location, the report of the explosion should presumably not appear. To handle this kind of situation we can use a SenseFuse:

```
new SenseFuse(obj, &prop, n, source, sense);
```

The two extra parameters are *source* and *sense*. With a SenseFuse `obj.prop` will still be executed after *n* turns, but anything that obj.prop tries to display to the screen won't actually appear unless the player character can sense *source* via *sense* (which must be one of sight, sound, smell or touch). We could revise our dynamite accordingly:

```
dynamite: Thing 'dynamite/stick' 'dynamite'
     dobjFor(Burn)
     {
         verify() {}
         action()
         {
              "You set the dynamite alight. ";
              new SenseFuse(self, &explode, 3, self, sound);
         }
     }
     explode()
     {
          "Bang! ";
          moveInto(nil);
     }
;
```

With this definition, if the dynamite is out of earshot when the fuse goes off, the dynamite is still moved out of play, but the "Bang!" message will not be displayed.

It should be added that although the dynamite example implements a fuse in a rather literal sense, Fuses and SenseFuses can be used to trigger any kinds of event we like.

If we want a repeating event rather than a one-off event, we use a Daemon rather than a Fuse. This is created in much the same way:

```
new Daemon(obj, &prop, n);
```

or

```
daemonID = new Daemon(obj, &prop, n);
```

This causes `obj.prop` to be executed every *n* turns. If n is 1, `obj.prop` is first executed on the current turn; if it is 2, it is next executed on the following turn (and so on).

For example:

```
cave: Room 'small cave'
     startDrip()
     {
         dripCount = 0;
         dripDaemon = new Daemon(self, &drip, 1);
     }
     stopDrip()
     {
         if(dripDaemon != nil)
         {
              dripDaemon.removeEvent();
              dripDaemon = nil;
         }
     }

     dripDaemon = nil
     dripCount = 0
     drip()
     {
        switch(++dripCount)
        {
           case 1: "A faint dripping starts. "; break;
           case 2: "The dripping gets louder. "; break;
           case 3: "The dripping becomes louder still. "; break;
           default: "There's a continuous loud dripping. "; break;
        }
     }
;
```

This code should be clear enough; note how the `stopDrip()` method checks that `dripDaemon` is not nil before attempting to call the `removeEvent()` method on it; this is a defensive programming strategy to ensure that it's always safe to call `stopDrip()` without causing a run-time error. The alternative would be to call:

```
eventManager.removeMatchingEvents(cave, &drip).
```

Corresponding to the SenseFuse is the SenseDaemon, defined in a similar way:

```
new SenseDaemon(obj, &prop, n, source, sense);
```

Here all the parameters have the meanings we've already seen. For example, in order to ensure that we only report the dripping sound when the player is in the cave to hear it, we might have set up the dripping daemon with:

```
dripDaemon = new SenseDaemon(self, &drip, 1, self, sound);
```

In addition to the Daemon and the SenseDaemon, there's a PromptDaemon, which is run every turn just before the prompt is displayed. This is set up simply with

```
new PromptDaemon(obj, &prop);
```

This will cause `obj.prop` to be executed every turn, just before the command prompt. The `OneTimePromptDaemon` is a PromptDaemon (set up in the same way) that executes just once and then disables itself. This can be useful when we want something to happen right at the end of the current turn; it can also be useful to set things up just before the first turn.

We can control the order in which Daemons and Fuses are executed by overriding their `eventOrder` property; the lower the number, the earlier the Event will execute. The default value is 100.

For more information, look up BasicEvent and its subclasses in the *Library Reference Manual*.

8.2 Coding Excursus 13 – Anonymous Functions

It is possible to create not only objects but functions dynamically; these are then *anonymous* functions. At its most general, the syntax is:

```
new function(args)  {  function body };
```

This returns a pointer to the function thus created, which we could use to call the function; for example:

```
local f = new function(x, y) { return x + y; };
local sum = f(1, 2);
```

When executed, this would result in `sum` being evaluated to 3.

An anonymous function is not restricted to containing a single statement; an anonymous function can be as long and complex as we like. But where an anonymous function does consist of a single statement, generally an expression to be evaluated and returned by the function, we can use a short form of the syntax. The following is equivalent to the anonymous function we just defined above:

```
local f = { x, y: x + y };
```

Note that with this short-form syntax, the list of arguments (if any) is followed by a colon, which in turn is followed by the expression that the anonymous function is to return. We do *not* use the keyword `return` in this short-form syntax, and we do *not* follow the expression (inside the short-form anonymous function) with a semi-colon. Attempting to use a semi-colon inside a short-form anonymous function will result in a compilation error.

It's perfectly legal to define an anonymous function that takes no arguments at all, for example:

```
local hello = {: "Hello World! " };
```

Subsequently executing `hello()` will then cause "Hello World!" to be displayed. As we shall see shortly, this kind of anonymous function definition can be particularly useful in EventLists.

Anonymous functions can refer to local variables and to the self object that in scope at the time they are created. For example, the following is perfectly legal:

```
someObj: object
    name = 'banana'
    doName()
    {
         local str = 'split';
         local f = { x: name + x + str };
         return f(' ');
    }
;
```

The doName() method would return 'banana split'.

At this point, you may well be thinking "This looks all very nice, but what useful purpose does it serve?" We'll be seeing some uses for anonymous functions later on in this chapter, but one common use is as the argument to some function or method. An anonymous function definition is an expression, returning a function pointer. It can thus be passed to a method or function that expects a function pointer as an argument. For example, we could define:

```
function countItems(lst, func)
{
     local cnt = 0;
     for(local i; i <= lst.length(); i++;
     {
         if(func(lst[i]))
            cnt++;
     }
     return cnt;
}
```

This function takes two arguments, a list (we'll say more about lists in the next Coding Excursus, a little further on in this chapter) and a function pointer. It returns the number of items in the list for which the function returns true (when called with a list item as its parameter). So, for example, we could call it with something like:

```
evens = countItems([1, 2, 3, 4, 5], {x: x % 2 == 0 } );
```

And this would return the number of even numbers in the list [1, 2, 3, 4, 5]. We could subsequently call it with:

```
clothingCount = countItems(me.allContents, {x: x.ofKind(Wearable) };
```

And this would return the total number of Wearable items carried, worn, or indirectly carried by the player character.

As we shall see, we don't actually need to define this particular function, since there's already an equivalent method defined on the List class, but that, too, uses anonymous functions in much this way.

For a fuller account of anonymous functions, see the chapter on 'Anonymous Functions' in Part III of the *System Manual*.

8.3 EventLists

We've seen how a Daemon can be used to make something happen each turn, but it's often useful to be able to define a list of events, one of which is to occur on each turn. We can do this with the **EventList** class. This defines an **eventList** property, which should contain a list of items (in the form [item1, item2, ... item*n*]).

The items in an EventList can be any of the following:

- A single-quoted string (in which case the string is displayed)
- A function pointer (in which case the function is invoked without arguments)
- A property pointer (in which case the property/method of the self object (i.e. the EventList object) is invoked without arguments)
- An object (which should be another Script or EventList), in which case its doScript() method is invoked.
- nil (in which case nothing happens).

Each item is dealt with in turn when the EventList's **doScript()** method is executed. We could thus use a Daemon to drive an **EventList** simply by repeatedly calling its **doScript()** method. For example, the dripping cave example could have been written:

```
cave: Room 'small cave'
     startDrip()
     {
         dripDaemon = new Daemon(self, &drip, 1);
     }

     stopDrip()
     {
         if(dripDaemon != nil)
         {
              dripDaemon.removeEvent();
              dripDaemon = nil;
         }
     }

     dripDaemon = nil

     drip() { dripEvents.doScript(); }

     dripEvents: EventList
     {
        eventList =
        [
          'A faint dripping starts. ',
          'The dripping gets louder. ',
          'The dripping becomes louder still. ',
          'There\'s a continuous loud dripping. '
```

```
       ]
    }
;
```

Actually, this is not *quite* the same, since an `EventList` stops doing anything at all when it runs off the end, whereas we want the “continuous loud dripping” message to keep repeating; for this we need a `StopEventList` rather than a plain `EventList`. Also, since we so often need to define the `eventList` property of an `EventList`, this can be done via a template. We could therefore define the `dripEvents` property as:

```
    dripEvents: StopEventList
    {
       [
         'A faint dripping starts. ',
         'The dripping gets louder. ',
         'The dripping becomes louder still. ',
         'There\'s a continuous loud dripping. '
       ]
    }
```

The various kinds of `EventList` we can use are:

- `EventList` – this runs through its eventList once, in order, and then stops doing anything once it passes the final item.
- `StopEventList` – this runs through its eventList once, in order, and then keeps repeating the final item.
- `CyclicEventList` – this runs through its eventList, in order, and returns to the first item once the final item is past.
- `RandomEventList` – this chooses an item at random each time it’s evoked.
- `ShuffledEventList` – this (usually) sorts the items in random order before running through it for the first time. It then runs through the items until it reaches the last one. After it’s used the last one it sorts the items in random order again and starts over from the beginning. The effect is a little like repeatedly shuffling a pack/deck of cards and dealing one at a time.
- `SyncEventList` – an event list that takes its actions from a separate event list object. We get our 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.
- `ExternalEventList` – a list 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).

We may often use `RandomEventList` and `ShuffledEventList` to provide atmospheric background messages (e.g. descriptions of various small animals and birds rustling around in a forest location). To prevent such messages out-repeating their welcome we can control their frequency with the properties **eventPercent**, **eventReduceTo** and **eventReduceAfter**. If we set **eventPerCent** to 75, 50, or 25, say, then a **RandomEventList** or **ShuffledEventList** will only trigger one of its items on average on three-quarters, or half, or one-quarter of the turns. If we want this frequency to fall after a while, we can specify a second frequency in **eventReduceTo** which will come into effect after we've fired events **eventReduceAfter** times. If we don't want the frequency to change, we should leave **eventReduceAfter** at nil. If we want this functionality on any other kind of EventList we can use the **RandomFiringScript** mix in class (e.g. we could define something as **RandomFiringScript, StopEventList**).

A **ShuffledEventList** has a couple of other properties we can use to tweak the way it behaves. If we *don't* want the events to be shuffled first time through (because we want them to be fired in the order we defined them first time round), we can set **shuffleFirst** to nil. If, on the other hand, we want a separate set of events to be triggered before we start on the shuffled list, we can define a separate **firstEvents** property. To allow us to define this easily, there's a **ShuffledEventList** template; if we define a **ShuffledEventList** with two lists (without explicitly assigning them to properties) , then the first list will be assigned to the **firstEvents** property, and the second to the **eventList** property, e.g.:

```
someList: ShuffledEventList
   ['First message', 'Second message' 'Third message' ]
   ['A random message', 'Another shuffled message',
      'Yet another shuffled message' ]
;
```

So far, all our examples have been of event lists containing single-quoted strings, but as we said at the outset, this is only one kind of item that can go there. We can't put a double-quoted string in an event list, even though we might want to (typically in order to take advantage of the **<< >>** embedded expression syntax), but we can put a function pointer in an event list, and such a function pointer could come from a short-form anonymous function containing a double-quoted string:

```
myList: EventList
  [
      'A single-quoted string. ',
      {: "A double-quoted string with an <<embeddedExpression()>>. " }
  ]
  embeddedExpression() { "embedded expression"; }
;
```

An alternative would be to use a property pointer:

```
myList: EventList
  [
      'A single-quoted string. ',
      &sayDouble
  ]
  sayDouble() { "A double-quoted string with an <<embeddedExpression()>>. "; }
  embeddedExpression() { "embedded expression"; }
;
```

This is more verbose in this case, but usefully illustrates how to use a property pointer in an EventList. In any case, we can use one syntax or the other to do something rather more complicated that display a double-quoted string, for example:

```
floorList: EventList
    [
        'The floor starts to creak alarmingly. ',
        'The creaking from the floor starts to sound more like cracking. ',
        new function()
        {
             "With a loud <i>crack</i> the floor suddenly gives way, and you
              suddenly find yourself falling...";
             gPlayerChar.moveIntoForTravel(cellar);
             gPlayerChar.lookAround(true);
        }
    ]
;
```

In this case, it’s a matter of individual preference whether we prefer to include an anonymous function within the list itself, or implement it as a separate method called via a property pointer:

```
floorList: EventList
    [
        'The floor starts to creak alarmingly. ',
        'The creaking from the floor starts to sound more like cracking. ',
        &floorBreak
    ]

    floorBreak()
    {
       "With a loud <i>crack</i> the floor suddenly gives way, and you
        suddenly find yourself falling...";
       gPlayerChar.moveIntoForTravel(cellar);
       gPlayerChar.lookAround(true);
    }
;
```

We can always drive an EventList by calling its doScript() method from a Daemon, but there are some places where the library provides hooks for EventLists that will be driven for us if we define them in the appropriate place.

For example, **Room** defines an **atmosphereList** property. If this is defined at all, it should be defined as an EventList of some kind, which will then automatically have its **doScript()** method called every turn the player character is in the room in question, for example:

```
class ForestRoom: OutdoorRoom
   atmosphereList: ShuffledEventList
   {
      [
         'A squirrel darts up a tree and vanishes out of sight. ',
         'A fox runs across your path. ',
         'You hear a small animal rustling in the undergrowth. ',
         'Some distance off to the right, a pair a birds take flight. '
      ]

      eventPercent = 80
      eventReduceTo = 40
      eventReduceAfter = 4
   }
;
```

Here we use a `ShuffledEventList` (probably the most suitable class for an atmosphere list), and reduce its frequency so that the player doesn't tire of our atmospheric messages too quickly.

Another place where an `EventList` is automatically useful is in conjunction with a `TravelMessage`. If the `TravelMessage` is also an `EventList`, then traversing this kind of `TravelConnector` (or any other `TravelConnector` that has `TravelWithMessage` in its class list) will automatically call its `doScript()` method (provided we haven't otherwise overridden its `travelDesc()` method). For example:

```
clearing: OutdoorRoom 'Forest Clearing'
    "It looks like you could go east or west from here. "
    west = streamBank
    east: TravelMessage, StopEventList
    {
         destination = roadSide
         eventList =
         [
             'You walk eastwards for several hundred yards down a track that
              seems to get narrower and narrower, until you're forced to
              squeeze through the tightest of gaps between trees. After that
              the track gradually widens out again, until you at last find
              yourself emerging by the side of a road. ',

             'You once again walk eastwards down the narrow track, squeeze
              through the gap, and emerge by the side of the road. '
         ]
    }
;
```

Here it might be tedious for the player to see the somewhat lengthy description of the walk down the track on each occasion, so we provide an abbreviated version for second and subsequent attempts.

We'll be meeting more of these built-in hooks for event lists in later on; in the meantime, for more information on the EventList classes look up the Script class and its subclasses in the *TADS 3 Library Reference Manual*. You might want to look up `ShuffledList` at the same time; although this is not a kind of EventList, there may be occasions when you'd find in useful (specifically when you want a sequence of values returned in a shuffled order, rather than a sequence of events executed in a shuffled order).

8.4 Coding Excursus 14 – Lists and Vectors

We've already mentioned Lists several times; the time has come to look at them a bit more closely. Since Vectors are quite similar to Lists , we may we well consider them together.

As we have already seen, a List is simply a series of values combined together as a single value. To define a constant list, we enclose the items in the list in square brackets and separate each element in the list from the next with a comma, e.g.:

```
local numlst = [1, 2, 3, 5, 7, 10];
local objlst = [redBall, greenBall, brownCow, blackShirt];
```

The above example assigns lists to local variables; they can equally well be assigned to object properties, or passed as arguments to functions or methods, or indeed returned as the value of a function or method, e.g.:

```
sumProduct(lst)
{
    local sum = 0;
    local prod = 1;
    for(local i = 1; i <= lst.length() ; i++)
    {
        sum += lst[i];
        prod *= lst[i];
    }
    return [sum, prod];
}
```

This function takes a list of numbers as its argument, and returns a list containing the sum and the product of these numbers. This demonstrates, among other things, how we can return more than one value from a function (or method) by using a list.

The example shows how we can get at the individual items in a list. To get at item `i` in list `lst` we simply give the list name followed by the index in square brackets: `lst[i]`. It's also legal to change a list value this way, e.g. `lst[4] = 15`. A list is indexed starting at 1, so that, for example, in the `numlst[1]` would be 1 and `objlist[1]` would be `redBall` (assuming these two lists are defined as shown above). The number of items in a list is given by its `length()` method, so that, for example `numlst.length()` would be 6 and `objlst.length()` would be 4. It's illegal to try to refer to a list element beyond the end of the list (e.g. an attempt to refer to `objlst[5]` would result in a run-time error, unless we'd made the list longer somehow).

All the lists we've seen so far have contained elements of the same type, but it's perfectly legal to mix datatypes in a list; the following, for example, is a perfectly valid list:

```
[1, 'red', greenBall, 5, &name]
```

It's also perfectly legal for a list to contain other lists as elements, for example:

```
local lst = [1, 3, ['red', redBall], [4, 5], 'herring'];
```

In this case `lst[3]` would yield the value `['red', redBall]` whereas `lst[3][2]` would yield the value `redBall`.

We can add elements to a list using the + operator. For example if `lst` is the list `[1, 3, 5]` then `lst + 7` would be the list `[1, 3, 5, 7]`. If we wanted to change `lst` to the list `[1, 3, 5, 7]` we could use the statement `lst += 7.`

We can also use the – operator to remove an item from a list. If `lst` were `[1, 3, 5, 7]` then `lst - 3` would be `[1, 5, 7]`; if we want to apply the change to `lst` (rather than assigning the changed list to another variable), we could do so with `lst -= 3`.

This raises an important point to remember: *using methods or operators on lists yields a new value which we can assign to something else, but does not in itself change the list operated on unless we explicitly make it do so*.

In other words, it's fatally easy to write an expression like `lst + 2;` when what we really needed was the assignment statement `lst += 2;`. The former is perfectly legal as a statement and will compile quite happily; it just won't do what we probably want.

For full details of how the + and – operators work with Lists and Vectors, see the chapter on 'Expressions and Operators' in Part III of the *TADS 3 System Manual*.

There are also quite a few methods of the List (and Vector) class that's it's useful to know about. We've already met one, `length()`; we'll now introduce a few more.

The `append()` method is quite similar to the + operator. That is `lst.append(x)` does much what `lst + x` does, namely adds x as a new element to the end of `lst`. Note, however, that this is an *expression*. Simply writing the statement `lst.append(x)` will *not* change the value of `lst`; instead it will return a new list that's `lst` plus `x`. If we want to change the value of `lst` using `append()` we need to write `lst = lst.append(x)`. There's also a subtle difference between + and `append()`. The difference is that `append()` always treats its argument as a single value, even if it's a list. The effect is that if lst is, say, [1, 3, 5] then:

```
lst + [7, 9] is [1, 3, 5, 7, 9]
lst.append([7,9]) is [1, 3, 5, [7, 9]]
```

Similar to `append()` is `appendUnique()`, except that each value in the combined list will appear only once, so for example:

```
[1, 2, 2, 4, 7].appendUnique([1, 3, 5, 7]) is [1, 2, 3, 4, 5, 7]
```

Related to `appendUnique()` is `getUnique()`, which simply returns a List containing each element only once.

```
[1, 2, 2, 4, 7].getUnique() is [1, 2, 4, 7]
```

The `countOf(val)` method returns the number of elements of the list equal to `val`, so for example `[1, 2, 2, 4, 7].countOf(2)` would return 2.

Similarly `indexOf(val)` returns the index of the first item in the list that's equal to `val`; so if `lst` is `[1, 2, 2, 4, 7]` then `lst.indexOf(2)` would be 2 while `lst.indexOf(7)` would be 5 and `lst.indexOf(3)` would be nil (showing that we can use `indexOf()` to test whether a list contains a particular value).

The two methods `countOf()` and `indexOf()` have a pair of powerful cousins called `countWhich()` and `indexWhich()`. The argument to these methods is an anonymous function, itself with one argument, which should return true for the condition we're interested in. For example, suppose that we've defined a Treasure class, and we want to know how many items of Treasure the player character is carrying (directly or indirectly). Using `countWhich()` we can do it like this:

```
  local treasureNum = me.allContents.countWhich({x: x.ofKind(Treasure) } );
```

Likewise if we want to identify which (if any) of the items the player is carrying is a `Treasure`, we can do so with this code:

```
   local idx = me.contents.indexWhich({x: x.ofKind(Treasure) } );
   local treasureItem = me.contents[idx];
```

In fact, we can do this even more compactly using the `valWhich()` method, which gives us the matching value directly, rather than its index position within the list.

```
   local treasureItem = me.contents.valWhich({x: x.ofKind(Treasure) } );
```

This kind of thing may look a little scary at first sight, but it's *well* worth getting used to, since it enables us to manipulate lists (and vectors) so economically. The alternative would be to write a loop:

```
    local treasureItem = nil;
    foreach(local cur in me.contents)
    {
        if(cur.ofKind(Treasure))
        {
             treasureItem = cur;
             break;
        }
    }
```

While this isn't too terrible, it's clearly quite cumbersome compared with using `valWhich()`, which enables us to achieve the same result in a single line of code. But it does, incidentally, introduce a new kind of loop, the `foreach` loop, which, as you may have gathered from the context, allows us to iterate over the elements of a List (or Vector). The general syntax should be reasonably apparent from the example:

```
   foreach(iterator-variable in list-name)
       loop-body
```

The idea is that *iterator-variable* takes the value of each element of *list-name* in turn, until we reach the end of the list (or encounter a `break` statement).

Among the other List methods available, we should mention `sublist()` and `subset()`, both of which provide means of extracting some group of elements from a list. The method `sublist(start, len)` returns a new list starting with the *start* element of the list we're operating with and continuing for at most *len* elements. The *len* argument is optional; if it's absent, we simply continue to the end of the list, so for example, if we have:

```
local a = [1, 2, 3, 4, 5];
local b = a.sublist(3);
local c = a.sublist(3, 2);
```

Then b will be `[3, 4, 5]` whereas c will be `[3, 4]`.

The `subset()` function takes an anonymous function as an argument, and returns a list of all the elements for which the anonymous function evaluates to true. For example, if we want a list containing all the `Treasure` items directly or indirectly in the player character's inventory we could generate it with:

```
me.allContents.subset({x: x.ofKind(Treasure)})
```

Alternatively, if we wanted a list of everything directly carried by the player with a bulk greater than 4, we could generate it with:

```
me.contents.subset({x: x.bulk > 4})
```

There are also methods to sort Lists, remove elements from Lists, and do various other interesting and useful things with Lists. For a full account, read the chapter on 'List' in Part IV of the *TADS 3 System Manual*.

One further function to be aware of is `nilToList()`; if the argument to this function is a list, the function returns the list unchanged, but if the argument is nil the function returns the empty list `[]`. This can be useful when we want to perform a list operation on a property that may contain either a list or nil. For example, suppose we want to add a precondition for an action on a particular object, we might write:

```
    preCond = inherited + objHeld
```

Should the object inherit from a class where the precondition for that action is undefined (and hence nil), this will cause a run-time error. We can avoid this danger by instead writing:

```
    preCond = nilToList(inherited) + objHeld
```

Apart from the **`nilToList()`** function, virtually everything we've said about Lists also applies to Vectors, so now we should say something about the difference between the two. The key difference is that a List is immutable while a Vector is not. That means that if we perform some operation on a List, we don't change the List, we create a new List with the revised set of values. A Vector, however, can be dynamically changed. This makes updating a Vector more efficient than updating a List, but also has implications for the effect of the change. As the *System Manual* explains it, if we defined the following:

```
    local a = [1, 2, 3];
    local b = a;
    a[2] = 100;
    say(b[2]);
```

When we display the second element of b we'll see the value 2 displayed. This is because when we change the second element of a we create a new List which is then assigned to a, but this does not affect the List that's assigned to b.

If, however, we attempted the equivalent operation with a Vector, we'd get a different result:

```
    local a = new Vector(10, [1, 2, 3]);
    local b = a;
    a[2] = 100;
    say(b[2]);
```

Displaying the second element of b would now show it to be 100. Since Vectors can be changed, no new object is created when we change the second element of the Vector, and so a and b continue to contain the same Vector object.

This example shows that creating a Vector is a bit different from creating a list. There's no such thing as a Vector constant equivalent to a List constant like **`[1 ,2, 3]`**. Vectors have to be created dynamically with the **`new`** keyword. The constructor can take one or two arguments. The first argument must be an integer specifying the initial allocation size of the Vector. So for example, we could create a Vector with a statement like:

```
     myProp = new Vector(20);
```

This would create a Vector with an initial memory allocation for 20 elements. This does not mean that the Vector is created with 20 elements; it is created empty. It also does not mean that the Vector is limited to 20 elements; we can carry on adding as many elements as we like. It simply means that we expect the Vector to grow to about 20 elements, and things will be a bit more efficient if our guess is more or less right.

We can also add a second argument, which can be either an integer or a List. With this statement:

```
myProp = new Vector(20, 10);
```

We'd create a new Vector and initialize it with 10 nil elements. With this one:

```
myProp = new Vector(20, [1, 3, 5]);
```

We'd create a new Vector and initialize its first three elements to 1, 3 and 5. This form of the Vector constructor effectively enables us to convert a List into a Vector (or, strictly speaking, to obtain a Vector containing the same elements as any given List). We can carry out the opposite operation with the `toList()` method, which returns a List containing the same elements a the Vector its called on. It can optionally return a subset of the elements from the Vector by specifying one or two optional arguments; `vec.toList(start, count)` will return a List containing *count* elements starting with the *start* element of `vec`.

Vector also defines many of the same methods we have seen for List, which do similar things, but with one important difference: *many methods that return a new List but leave the original List unchanged* will *change a Vector when executed on a Vector*.

If we want an object property to hold a Vector, there's a couple of ways we can typically go about it. One coding pattern is to start with a nil value and to create the Vector dynamically the first time we try to add an element to it:

```
myObj: object
   vecProp = nil
   addVecElement(val)
   {
       if(vecProp == nil)
           vecProp = new Vector(25);

       vecProp.append(val);
   }
;
```

The alternative is to assign a Vector to the property's initial value using the `static` keyword:

```
myObj: object
    vecProp = static new Vector(25)
;
```

We'll say more about the `static` keyword below.

The main question this all leaves is why one should use a Vector in preference to a List. The answer is that it's more efficient to change a Vector than a List (the latter requiring the overhead of creating a new object each time it's updated). It can therefore lead to better performance if we use a Vector for properties that are likely to be changed frequently, or when building a set of values dynamically. In the latter case we can always convert the Vector to a List once we've built it.

This section has introduced only the most salient features of Lists and Vectors. For the full story see the 'List' and 'Vector' chapters in Part IV of the *TADS 3 System Manual*.

8.5 Initialization and Pre-initialization

8.5.1 Initialization

We've seen how we can use Daemons and Fuses to trigger certain kinds of events, and EventLists to control sequences of Events; one other place where we might want to make things happen is when our game starts.

One way we can do that is with an `InitObject`. An `InitObject` is simply an object whose `execute()` method will be executed when the game starts up. `InitObject` can be mixed in with other classes so that an object's initialization code can be written on the object. This can be particularly useful for starting a Fuse or Daemon at the start of play, for example:

```
bomb: InitObject, Thing 'long black cylinder/bomb*bombs'  'bomb'
    "It looks like a long black cylinder. "
    execute()  {   fuseID = new Fuse(self, &explode, 20); }
    fuseID = nil
    explode()
    {
         "The bomb explodes with a mighty roar! ";
          if(gPlayerChar.isIn(getOutermostRoom))
               gPlayerChar.die();

          moveInto(nil);
    }
;
```

If we want, we can control the order of execution through the `execBeforeMe` and `execAfterMe` properties. These properties can hold lists of InitObjects that should be executed before or after the InitObject we're defining. For example, if we went on to define a second InitObject we wanted to execute after the bomb sets up its Fuse, we'd define `execBeforeMe = [bomb]` on it.

8.5.2 Pre-Initialization

Initialization takes place at game start-up. Pre-Initialization takes place towards the end of the compilation process; we can therefore use it to set up data structures and carry out calculations that need to be in place at the start of play, without causing any delay at the start of play, since the result of these calculations will be part of the compiled game image.

Just as we can use `InitObjects` to carry out tasks at initialization, so we can use

PreinitObjects to carry out tasks at pre-initialization.

Apart from the stage at which its **execute()** method is executed, a **PreinitObject** works in much the same way as an **InitObject**. As with InitObjects, we can define as many PreinitObjects as we like, mix **PreinitObject** in with other classes, and use its **execBeforeMe** and **execAfterMe** properties to control the order in which PreinitObjects are executed.

For example, suppose at various points in our game we want to check the status of all objects belonging to our custom **Treasure** class. To do that, it would be helpful to have a list of them all stored somewhere; we could build it using a **PreinitObject** thus:

```
treasureManager: PreinitObject
    treasureList = []
    execute()
    {
        for(local obj = firstObj(Treasure); obj != nil;
                                                      obj = nextObj(obj, Treasure))
               treasureList += obj;
    }
;
```

If we then need to iterate over all the **Treasure** objects in the course of play, we can then do so using the list in **treasureManager.treasureList.** (We'll be properly introduced to the **firstObj()** and **nextObj()** methods in the next chapter).

The existence of both **InitObject** and **PreinitObject** raises the question of which to use when. The general rule is probably to use **PreinitObject** wherever possible, and **InitObject** otherwise. Situations in which we have to use an **initObject** rather than a **PreinitObject** include:

- Outputting text to the screen, or accepting input from the player.
- Creating Fuses and Daemons.
- Testing the capabilities of the interpreter the game is running on (e.g. with the **systemInfo()** function), and setting things up accordingly.
- Setting up something random.

For the full story on Initialization and Pre-Initialization see the chapter on 'Program Initialization' in Part V of the *TADS 3 System Manual*.

8.5.3 Static Property Initialization

This seems a convenient point to mention one other means of carrying out useful calculations at compile time, namely static initialization. This is actually carried out just before pre-initialization, and allows us to assign an expression to an object property at compile time by using the `static` keyword. This expression can, for example, be an object that has to be created dynamically, such as a Vector, e.g.:

```
   agendaList = static new Vector(15)
```

As another example, we might want to set the `reduceEventAfter` property of a `ShuffledEventList` to the number of items in its `eventList` property, since it would make good sense to reduce the frequency of random atmospheric messages once the player has seen every one of them once. We could do this with:

```
    eventReduceAfter = (eventList.length())
```

This would have the advantage that `eventReduceAfter` would contain the right value even if we decide to add more atmosphere strings to the `eventList`. But it would be more efficient to use static initialization:

```
    eventReduceAfter =  static eventList.length()
```

Since with this code, the length of the `eventList` is calculated at compile time and the value is assigned to the `eventReduceAfter` property as a constant value.

Any valid expression may follow the `static` keyword. For a fuller account, see the ‘Static Property Initialization’ section of the ‘Object Definitions’ chapter in Part III of the *TADS 3 System Manual*.

Exercise 15: Try creating the following game. The player character starts in a living room in wartime London, in which an unexploded bomb lies on the floor. He has 25 turns in which to defuse the bomb, after which it will explode. To defuse the bomb he has to remove a metal cap from it, which can only be removed with the aid of his spanner. This is one of his tools, which starts out in his black tool bag, which is out in the hall. The tool bag also contains his wire cutters and his bomb disposal manual.

Removing the cap from the bomb reveals five coloured wires in the detonator: red, blue, green, yellow, black. Cutting the right wire will defuse the bomb, but cutting the wrong one will make it go off. To find out which is the right wire, the player must determine which kind of bomb it is and then look it up in the bomb disposal manual. The serial number of the bomb is on the underside of the casing, so the player must look under the bomb to find it.

Out in the hall an inquisitive rat is scurrying around, so we should display a series of messages describing what it’s up to. We should also display a series of random messages describing sounds coming from outside the house. Finally, when there’s only five turns left, the bomb should start ticking louder, as a hint to the player that he needs to hurry up.

We'll add some further finishing touches to this game in the next chapter, but in the meantime you might like to compare your version with the Bomb Disposal sample game.

9 Beginnings and Endings

9.1 GameMainDef

Most games start with some kind of introductory text before the first room description, to set the scene and maybe give some brief instructions to the player. The normal place to do this in the `showIntro()` method of the `gameMain` object, which we have to define for every TADS 3 game:

```
gameMain: GameMainDef
   showIntro()
    {
        "Welcome to Zork Adventure, a totally original treasure-hunt set in
         the Colossal Underground Cave Empire. Armed only with a bottle
         of water, your trusty carbide lamp, and your wits, you must overcome
         a small army of dwarvish grues and gruesome dwarves to collect the
         famed fifty-five firestones of Fearsome Folly!\b
         First time players should type ABOUT. ";
    }
;
```

Although we don't absolutely *have* to have a `showIntro()` method, it's generally a good idea, and we do absolutely have to have a `gameMain` object which must be of the `GameMainDef` class.

The one property of `gameMain` we absolutely *must* define is `initialPlayerChar`, which defines which object represents the player character at the start of play. In most TADS 3 games (especially those created from the starter game templates used by Workbench) the initial player character is called `me` (though we could call it anything we liked), so a minimal `gameMain` would typically consist of:

```
gameMain: GameMainDef
    initialPlayerChar = me
;
```

Previous chapters have sometimes referred to the player character as me, and sometimes as `gPlayerChar`; a word of explanation is now in order. `gPlayerChar` is a macro defined as:

```
#define gPlayerChar (libGlobal.playerChar)
```

In the `adv3LibPreinit` object (a `PreinitObject`) the following statement occurs:

```
gPlayerChar = gameMain.initialPlayerChar;
```

In other word `gPlayerChar` (aka `libGlobal.playerChar`) is pre-initialized to the value of `gameMain.initialPlayerChar`, which is usually `me`. If the player character remains the same throughout the game, as if often, if not usually, the case, then

gPlayerChar and **me** will refer to the same object throughout the game. It's then a matter of preference which of these we use to refer to the player character, although **me** is generally quite a bit less typing! If, however, we're writing a game in which the player character changes (or may change) in the course of play, it's probably best to use **gPlayerChar** to refer to the current player character throughout.

We can override a number of other properties on **gameMain**, but most of these are either ones that are best dealt with in later chapters as they become relevant, or left for the reader to investigate in due course. A few of the more commonly useful properties of **gameMain** include:

- **allVerbsAllowAll** – by default this is true, but if we set it to nil this restricts the use of ALL to a handful of inventory-handling verbs. This prevents the player from taking a brute force approach to game-play and puzzle-solving by disallowing commands like EXAMINE ALL and SHOW ALL TO BOB.
- **beforeRunsBeforeCheck** – changes the order of the before() and check() handling; we'll come back to this in the 'More About Actions' chapter.
- **cancelCmdLineOnFailure** – this is nil by default; if it's set to true then if the player enters multiple commands at once (e.g. **north, north, take ball, hit troll, west**) and one of them fails, the remainder are ignored.
- **usePastTense** – this is nil by default, but if it's set to true all the library messages are displayed in the past tense (for use in a game narrated in the past tense). For more information on this, see the article on 'Writing a Game in the Past Tense' in the *TADS 3 Technical Manual* (but this can be left until you actually want to use the past tense).

There's also a couple of methods of **gameMain** we might want to use quite commonly. One is **showGoodbye()**, which can be used to display a parting message right at the end of the game, and **setAboutBox()** which can be used to set up an about box that displays when players use the Help->About option in their interpreter. This might typically look something like this:

```
gameMain: GameMainDef
   initialPlayerChar = me
   setAboutBox()
   {
      "<ABOUTBOX><CENTER>
       <b>ZORK ADVENTURE</b>\b
       Version 1.0\b
       by Watt A. Ripov
       </CENTER></ABOUTBOX>";
   }
;
```

Or you could make a general purpose about box that takes all its information from the **versionInfo** object:

gameMain: GameMainDef

```
   initialPlayerChar = me
   setAboutBox()
   {
      "<ABOUTBOX><CENTER>
       <b><<versionInfo.name>></b>\b
       Version <<versionInfo.version>>\b
       <<versionInfo.byline>>
       </CENTER></ABOUTBOX>";
   }
;
```

Such an automated about box has the merit that it will always accurately reflect changes made to versionInfo (which we'll look at more closely in just a moment), so that such changes only need to be made in one place.

For more information on `GameMainDef`, look up `GameMainDef` in the *Library Reference Manual*.

9.2 Version Info

The other object we have to define, along with `gameMain`, is `versionInfo`. This provides information about the name, version and author of our game, and can contain additional information for classifying our game. A typical versionInfo object may look like:

```
versionInfo: GameID
    IFID = '2b5c2e11-003f-6e0c-4d75-6092f703208b'
    name = 'Bomb Disposal'
    byline = 'by Eric Eve'
    htmlByline = 'by <a href="mailto:eric.eve@hmc.ox.ac.uk">
                  Eric Eve</a>'
    version = '0.1'
    authorEmail = 'Eric Eve <eric.eve@hmc.ox.ac.uk>'
    desc = 'A demonstration of Fuse and Daemon classes (and also InitObject,
        PreinitObject, CollectiveGroup and Consultable).'
    htmlDesc = 'A demonstration of Fuse and Daemon classes (and also InitObject,
        PreinitObject, CollectiveGroup and Consultable).'
;
```

This should be reasonably self-explanatory, but for further information look up `GameID` in the *TADS 3 Library Reference Manual* and read the article on 'Bibliographic Metadata' in the *TADS 3 Technical Manual*.

There are two methods we may well wish to define on this object, `showAbout()` and `showCredits()`. These methods define what is displayed in response to an **about** or **credits** command respectively, and should always be defined in a reasonably polished game to give appropriate responses. What we put in `showAbout()` can be anything from a brief set of instructions to an explanation of what the game is about with details of special commands and the like, to a full set of help menus. The response to `showCredits()` should normally acknowledge the assistance of anyone who has contributed to our game, including the authors of any extensions we have used and a list of beta-testers.

For further details of these two methods, look up **ModuleID** in the *Library Reference Manual*.

9.3 Coding Excursus 15 – Intrinsic Functions

TADS 3 defines a number of *intrinsic functions,* functions built in to the system. These are fairly fully documented in three of the chapters in Part IV of the *TADS 3 System Manual:* 't3vm Function Set', 'tads-gen Function Set' and 'tads-io Function Set'. Here we'll just take a brief look at some of the more generally useful ones. Most of these will be from the tads-gen Function Set.

One such function that we have already met is **dataType(val)**, which returns the data type of its *val* argument; see section 7.2 above.

Another pair of functions that we've met briefly are **firstObj()** and **nextObj()**. Together, these can be used to iterate over all objects in the game, or all objects of a certain class in the game; **firstObj(*cls*)** returns the first object of class *cls*; **nextObj(obj, cls)** returns the next object of class *cls* after **obj**. So, for example, to iterate over every object of class **Decoration** in the game (suppose, for some reason, we wanted to count the number of **Decoration** objects our game contained), we could use the two functions in tandem, either with:

```
local decorationCount = 0;
local obj = firstObj(Decoration);
while(obj != nil)
{
    decorationCount++;
    obj = nextObj(obj, Decoration);
}
```

Or, a little more succinctly, with:

```
local decorationCount = 0;
for(local obj = firstObj(Decoration); obj ; obj = nextObj(obj, Decoration) )
    decorationCount++;
```

Or, if we want to use a library-defined function **forEachInstance()**, which does part of this job for us, simply:

```
local decorationCount = 0;
forEachInstance(Decoration, {x: decorationCount++ } ) ;
```

Another useful pair of functions are **max()** and **min()** which return respectively the maximum and minimum value in their argument lists (which accordingly must contain values all of the same type). For example **max(1, 3, 5, 7, 9)** would return 9, while **min(1, 3, 5, 7, 9)** would return 1 (obviously this is rather more useful when the argument list contains at least one variable!).

The `rand()` function can be used both to return random numbers and to make random choices.

`rand(n)` (where n is an integer) returns a random integer between 0 and n-1. For example, `rand(10)` returns a random number between 0 and 9.

`rand(lst)`, where *lst* is a list, randomly returns one of the elements of lst.

`rand(val1, val2, ... valn)` (i.e. where `rand()` has two or more arguments) randomly returns one of the arguments.

The `randomize()` function is used to re-seed the random number generator (to ensure that we get a different sequence of random choices each time the program is run).

We would typically call `randomize()` right at the start of our game; if we do so it must be called in an `InitObject` (or `gameMain.showIntro()`) rather than a `PreinitObject`; the same applies if we want to do anything random at start up (e.g. making a random choice of what the safe combination will be, or of which wire to cut to disable a bomb).

Another pair of generally useful intrinsic functions are `toInteger()` and `toString()`. The first of these, `toInteger(val)` returns the value of val as an integer if *val* is an integer, a `BigNumber` within the integer range (-2147483648 to +2147483647) or a string comprising of digits (possibly with leading spaces, a leading + or a leading -). If *val* is true or nil the function returns true or nil respectively. Otherwise a runtime error is generated.

The second, `toString(val)`, returns a string representation of *val* if val is an integer, `BigNumber`, string, true or nil. Otherwise it throws an error.

The functions in the tads-io set are less generally useful to game authors than might generally appear. The `systemInfo()` function is useful if we want information about the interpreter and operating system our game is running on (e.g. to test whether it has graphical or HTML capabilities); for details see the description of this function in the 'tads-io Function set' chapter of the *System Manual*.

The various bannerXXX functions look interesting, but are quite tricky to use in practice. If you want banner functionality (the ability to divide the interpreter window up into sub-windows) in your game, consult the article on 'Using the Banner API' in the *Technical Manual*.

The functions `morePrompt()` (for pausing output and asking the player to press a key), `clearscreen()` (for clearing the screen), `inputKey()` (for reading a single keystroke) and `inputLine()` (for reading a line of text input by the player), all look as if they should do something useful, but for various reasons they probably won't do quite what we want or expect; it's better to avoid them and use the alternatives suggested below:

- Instead of `morePrompt()` use `inputManager.pauseForMore(true)`;
- Instead of `inputKey()` use `inputManager.getKey(nil, nil)`;
- Instead of `inputLine()` use `inputManager.getInputLine(nil, nil)`;
- Instead of `clearscreen()` use `cls()`;

For a fuller explanation of this, see the article 'Some Common Input/Output Issue' in the *TADS 3 Technical Manual*. The brief story is that TADS 3 buffers output through something called the *transcript,* part of the effect being that the raw intrinsic functions won't generally do their input and output at the exact point we expect them to; the methods of `inputManager` take the transcript into account, so that things will work as expected.

9.4 Ending a Game

We've now seen several ways to do things at the beginning of a game, but we've not yet seen how to bring a game to an end.

The normal way to end a TADS 3 game is to call the function `finishGameMsg()`. This is called with two arguments: `finishGameMsg(msg, extra)`. The first parameter, *msg*, can be a single-quoted string or a `FinishType` object. If it's a single-quoted string, this will be displayed as the game ending message, preceded and followed by three asterisks in the standard IF format; for example, calling `finishGameMsg('All Over', [])` will end the game displaying:

```
*** All Over ***
```

Alternatively the *msg* parameter can be a `FinishType` object, which can be used to display one of the very common ending messages:

- `ftDeath` – You have died
- `ftFailure` – You have failed
- `ftGameOver` – Game Over
- `ftVictory` – You have won

If there was some other message you thought you were going to use frequently, it would be very easy to define your own `FinishType` object, e.g.

```
ftWellDone: FinishType  finishMsg = 'Well Done!' ;
```

The second parameter, *extra*, should contain a list (which may be an empty list) of `FinishOption` objects. When the game ends the player is always offered the QUIT, RESTART and RESTORE options; the *extra* parameter defines which additional

FinishOptions the player is offered. The library defines the following **FinishOption** objects:

- **finishOptionAmusing** – offer the AMUSING option
- **finishOptionCredits** – offer the CREDITS option
- **finishOptionFullScore** – offer the FULL SCORE option
- **finishOptionQuit** – offer the QUIT option
- **finishOptionRestart** – offer the RESTART option
- **finishOptionRestore** – offer the RESTORE option
- **finishOptionScore** – offer the SCORE option
- **finishOptionUndo** – offer the UNDO option

As just noted, the QUIT, RESTART and RESTORE options are always offered as standard, so there’s no need to specify any of these in the *extra* parameter. The difference between the Full Score and the Score options is that the former displays a lists of achievements that make up the score, whereas the latter simply displays the final score.

So, for example, we might end the game with:

```
finishGameMsg(ftVictory, [finishOptionUndo, finishOptionFullScore]);
```

The game will then be ended with the message "*** You have won ***", following which the player will be offered the RESTART, RESTORE, FULL SCORE, UNDO and QUIT options.

If we include the AMUSING option, we also need to define what it does. To do this we need to modify **finishOptionAmusing** and override its **doOption()** method. This can then do whatever we like, but at the end it should normally return true in order to redisplay the list of options. For example:

```
modify finishOptionAmusing
    doOption()
    {
        "Have you tried asking Attila the Hun about his favourite opera,
         or drinking from the bottle marked <q>Cat Poison</q>, or riding
         the sea-horse, or smelling the drain in the sewerage room?\b";

         return true;
    }
;
```

Of course the AMUSING option could do something much more sophisticated than this, up to and including displaying a menu of sub-options.

It would also be possible, though seldom ever necessary, to define a FinishOption of your own. The following example illustrates the principle:

```
finishOptionBoring: FinishOption
    desc = "see some truly <<aHrefAlt('boring', 'BORING', '<b>B</b>ORING',
            'Show some boring things to try')>> things to try"

    responseKeyword = 'boring'
    responseChar = 'b'

    doOption()
    {
       "1. When play begins, try pressing Z exactly 1,234 times.\n
        2. Try climbing the north wall of the sitting room.\n
        3. Ask every NPC you meet everything you can think of about the
           first ten Roman emperors.\n
        4. Establish just how many doors, boxes and containers the bent
           brass key <i>won't</i> unlock.\b";

       return true;
    }
;
```

For more details (should you wish to create your own `FinishOption` and actually need more details), look up `FinishOption` and the various `finishOptionXXX` objects in the *Library Reference Manual*.

Exercise 16: Complete the Bomb Disposal example from Exercise 15 by adding appropriate introductory text and suitable winning and losing endings. Also add start-up code to randomize which is the correct wire for the player to cut.

10 Darkness and Light

10.1 Dark Rooms and Light Levels

As we've already seen, we can use the `DarkRoom` class to define a rooms that are dark unless the player manages to provide a light source. We can make other kinds of Room, such as `OutdoorRoom`, equally dark by setting their `brightness` property to 0. When the player character enters such a dark place, the player will see it described thus:

In the dark
It's pitch black!

If this isn't quite what we want, it's easy enough to customize. We can override the `roomDarkName` and `roomDarkDesc` to change the way the name and the description of the dark room is shown. For example, if the player character descends a flight of stairs into what's obviously a cellar (even though it's dark), it might be better if both the room name and its description reflected that:

```
cellar: DarkRoom 'Cellar'
   "This cellar is relatively cramped, with most of the space taken up by
    the rusty old cabinet in one corner and the pile of junk in the other.
    A flight of stairs leads back up. "

    roomDarkName = 'Cellar (in the dark)'
    roomDarkDesc = "It's too dark to make out much in here apart from the
     dim outline of the stairs leading back up out of the cellar. "

    up = cellarStairs
;
```

Then, when the player character enters the darkened cellar, it would appear as:

Cellar (in the dark)
It's too dark to make out much in here apart from the dim outline of the stairs leading back up out of the cellar.

This is fine, but we've now given ourselves another problem: we've mentioned the stairs leading back up out of the cellar, but while the cellar is in darkness the player won't be able to interact with them, either to examine them or to climb them (both of which would be perfectly reasonable actions under the circumstances). The simplest solution is to give the staircase object a `brightness` of 1. According to the comment in the *Library Reference Manual* a `brightness` of 1 means that the "object is self-illuminating, but doesn't give off enough light to illuminate other objects. This is suitable for something like an LED digital clock." In practice it's equally suitable for any object we want to be visible in an almost-dark place, for example a dimly-lit exit or bulky furniture.

It may be that we'd want such objects described differently (specifically in response to **examine**) when the room is dark from when it is light. To that end we need to know how light or dark the room is. We can't just test the brightness property of the Room, since that won't tell us whether the player is carrying a lamp, or whether there's a candle burning nearby, or whether there's some other source of light. The most general way to test the light level somewhere is to use the `senseAmbientMax()` method. This is called with one argument, which is a list of the senses we're interested in. To be absolutely copper-bottomed safe we could call it with `gPlayerChar.sightLikeSenses` as the argument, but in 99% of games it'll almost certainly be perfectly safe just to use `[sight]`. We could call the method on any object within visible range (such as the room itself, or the actor, or some object that would be in plain sight in the room if the room were lit); but we may as well call it on the object we're interested in. For example, in the case of the flight of stairs leading out of the cellar, we might define:

```
+ cellarStairs: StairwayUp 'dim outline/flight/stairs' 'flight of stairs'
  desc()
  {
    if(senseAmbientMax([sight]) > 1)
       "The stairs look well worn, but solid enough. ";
    else
       "It's just a dim outline in the dark; you as much sense as see
        that there's a flight of stairs there, and that only because
        you've just come down them. ";
  }

  brightness = 1
;
```

An alternative to giving the `cellarStairs` (or other such objects) a `brightness` of 1 is to use the `getExtraScopeItems()` method to add them to scope. The standard library already uses this to put the floor of a dark room in scope, so we must remember to add any additional objects to the inherited behaviour. For example, we could add the stairs to scope in the dark like this:

```
cellar: DarkRoom 'Cellar'
   "This cellar is relatively cramped, with most of the space taken up by
    the rusty old cabinet in one corner and the pile of junk in the other.
    A flight of stairs leads back up. "

    roomDarkName = 'Cellar (in the dark)'
    roomDarkDesc = "It's too dark to make out much in here apart from the
     dim outline of the stairs leading back up out of the cellar. "

    up = cellarStairs

    getExtraScopeItems(actor) { return inherited(actor) + cellarStairs; }
;
```

This behaves a little differently from giving the `cellarStairs` a `brightness` of 1. It should still allow the player character to climb the stairs, but an attempt to examine them will be met with the response "It's too dark to do that". Neither method is necessarily better than the other, it depends what effect we want, but if we're implementing a room that's in near-total darkness, and we don't particularly want to provide a separate description for the objects we want to be in scope when it's dark, `getExtraScopeItems()` may be the way to go.

One other action that behaves differently in the dark is moving around. The library applies the convention that a `TravelConnector` is visible in the dark if and only if its destination is lit (if I'm standing in a dark room I should be able to see the door if there's light on the other side of it). Moreover, when the player character attempts to move from one dark location to another, this is generally disallowed. This behaviour is enforced on the `darkTravel()` method of `TravelConnector`, which in turn calls the `roomDarkTravel()` method of the actor's location. This is defined on `BasicLocation`, and by default it simply calls `cannotGoThatWayInDark()` to display a message saying why travel in the dark is prohibited followed by `exit` to stop the action. We can thus override any of these methods if we wish, either to allow dark-to-dark travel through a specific connector, all to allow dark-to-dark travel in a specific location, or to change the message that's displayed when dark travel is not allowed. For example, we might do this:

```
modify DarkRoom
   cannotGoThatWayInDark()
   {
       "You'd better not go blundering around in the dark; you might be eaten
        by a grue! ";
   }
;
```

Or, if we were feeling a bit meaner:

```
class UndergroundRoom: DarkRoom
   roomDarkTravel()
   {
       "Blundering around in the dark is a perilous business. You are eaten
        by a grue!<.p>";
        finishGame(ftDeath, [finishOptionUndo, finishOptionFullScore]);
   }
;
```

Finally, since we gave the library's definition of a brightness level of 1, for the sake of completeness we should perhaps give its definition of all the brightness levels:

0: The object is giving off no light at all.

1: The object is self-illuminating, but doesn't give off enough light to illuminate any other objects. This is suitable for something like an LED digital clock.

2: The object gives off dim light. This level is bright enough to illuminate nearby objects, but not enough to go through obscuring media, and not enough for certain activities requiring strong lighting, such as reading.

3: The object gives off medium light. This level is bright enough to illuminate nearby objects, and is enough for most activities, including reading and the like. The intervention of an obscuring medium reduces this level to dim (2).

4: The object gives off strong light. This level is bright enough to illuminate nearby objects, and the intervention of an obscuring medium reduces it to medium light (3).

10.2 Coding Excursus 16 – Adjusting Vocabulary

We'll shortly be looking at a number of ways of providing light. Many light sources can be either lit or unlit; when lit the player should be able to refer to them as 'lit'; when unlit the player should be able to refer to them as 'unlit'. We therefore need some mechanism for adjusting an object's vocabulary during the course of play. This excursus will explore three ways of doing it.

10.2.1 Adding Vocabulary the Easy Way

As we've seen, we can define the initial vocabulary of an object (the words the player can use to refer to the object) by assigning it to the object's `vocabWords` property. This takes the form of a list of adjectives, separated by spaces, followed by a list of nouns, separated by slashes, and a list of plurals separated by asterisks (or starting with an asterisk). For example 'quick brown fox/animal*fox*foxes'. But what happens if we need to change an object's vocabulary during the course of play? If the fox is killed we might want to add 'dead' to its vocabulary; if a vase were dropped on the floor we might want to add 'broken' to its vocabulary; when the tall dark stranger eventually introduces himself we might want to add 'bob' to his vocabulary; how can we go about it?

One thing we can't do is simply change the `vocabWords` property at run-time, or the underlying `noun` and `adjective` properties. At least, we can change them, but it won't do any good; once the game is running we're beyond the point where the library does anything with these properties.

The easiest way to add new vocabulary to an object is by calling its `initializeVocabWith()` method. This takes one argument, a string in the same format as that used for `vocabWords`. If we want to add only adjectives, and no nouns or plurals, we can end the string with a dash (-) in what would otherwise have been the slot for the noun (or nouns).

So, for example, we could add the desired extra vocabulary in the three cases above by calling `fox.initializeVocabWith('dead -');` `vase.initializeVocabWith('broken -');` and `bob.initializeVocabWith('bob')`. Note that calling initializeVocabWith() does not remove any existing vocabulary from an object (the vocabulary in the argument is added to that for the object; it does not replace it).

10.2.2 Dictionary

The second way we can adjust vocabulary at run time is through manipulating the game's dictionary. This will be of the intrinsic class `Dictionary`, and will usually be called `cmdDict`. The two methods most useful for manipulating a `Dictionary` are:

- `addWord(obj, str, vocabProp)` – adds the string *str* to the dictionary as a reference to the object *obj* associated via *vocabProp* (`&noun`, `&adjective`, `&plural`, etc.). *str* can also be a list of strings. If we try to add a combination that already exists, it'll simply be ignored.
- `removeWord(obj, str, vocabProp)` – removes the association of *str* from *obj* as *vocabProp*, where these parameters have the same meaning as in `addWord()`. If the word association defined by the parameters does not exist in the dictionary, calling this method has no effect.

For example, an alternative way of adding the vocabulary in our previous examples would be to call `cmdDict.addWord(fox, 'dead', &adjective); cmdDict.addWord(vase, 'broken', &adjective);` and `cmdDict.addWord(bob, 'bob', &noun)`. If the fox had previously been described as 'lively' we might also want to call `cmdDict.removeWord(fox, 'lively', &adjective)`.

For more information on the Dictionary class, see the Dictionary chapter in Part IV of the *TADS 3 System Manual*.

10.2.3 ThingState

Some kinds of object, such as light sources which can be either lit or unlit, may switch states quite frequently. In such cases we may want particular vocabulary (such as 'lit' and 'unlit') associated with particular states. We could write code to add and remove words from the dictionary each time such objects change state, but this is perhaps a little cumbersome, and the library provides a neater mechanism for handling such cases, the `ThingState` class.

When an object can be in one of several states, we can define these states as `ThingState` objects. For example, for light sources the library defines `lightSourceStateOff` and `lightSourceStateOn`. To associate these states with an object or class of object, we list all the possible states of that object in its `allStates` property, and its current state in its `getState` property, for example:

```
class LightSource: Thing
   allStates = [lightSourceStateOff, lightSourceStateOn]
   getState = (isLit ? lightSourceStateOn : lightSourceStateOff)
   isLit = nil
;
```

We can then define the state-specific vocabulary words relating to particular states in the `stateTokens` property of the relevant `ThingState` objects. This should hold a list of strings defining vocabulary words that will only be recognized for an object (such as a **LightSource**) when it's in this state. For example:

```
lightSourceStateOff: ThingState
    stateTokens = ['unlit']
;

lightSourceStateOn: ThingState
    stateTokens = ['lit']
;
```

For this to work, 'lit' and 'unlit' must *also* be included in the **vocabWords** of the objects to which these ThingStates apply (in this case the **LightSource** class). In other words, ThingStates do *not* provide a method for *adding* new vocabulary to an object; they instead provide a mechanism for *filtering* the existing vocabulary (and so, for example, the library defines the **LightSource** class as having the adjectives 'lit' and 'unlit').

But this is not the only mechanism **ThingState** provides. It also provides a mechanism for providing extra pieces of parenthetical information, like "(providing light)", added to the name of the item in inventory listings and the like. The **listName()**, **inventoryName()**, and **wornName()** methods define the additional information to be added to the name of the item in room/contents listings, inventory listings, and listings of items worn by the actor respectively. Each of these methods take a single parameter, *lst*, which contains a list of the objects concerned; this will either be a list containing a single element (the item to be listed), or a list of equivalent items. By default **inventoryName()** and **wornName()** return the value of **listName()**, while the **listName()** method returns the value of the **listName_** property (note the underscore). This allows a **ThingState** to be initialized via a template (**listName_** being the property assigned a value via the single-quoted string in the template). Thus, for example, the full definition of the two ThingStates mentioned above is:

```
lightSourceStateOn: ThingState 'providing light'
    stateTokens = ['lit']
;
lightSourceStateOff: ThingState
    stateTokens = ['unlit']
;
```

The library only defines half a dozen of these **ThingState** objects; two for use with the **LightSource** class, two for use with the **Matchstick** class; and two for use with the **Wearable** class. There is, however, nothing to stop us defining ThingStates of our own, and no reason why we shouldn't define them for individual objects if it's useful to do so, giving them whatever additional methods or properties we choose.

For further details, look up **ThingState** in the *Library Reference Manual*.

10.3 Sources of Light

If we define one or more dark rooms in our game, the chances are we're expecting our players to find some way of bringing light to them. Our next task, then, is to look at the various ways the TADS 3 library provides support for this.

The most basic way of providing a light source in TADS 3 would be to change the brightness of some Thing to 2, 3 or 4. So, for example, we might define:

```
magicCrystal: 'magic glowing eerie light/crystal*crystals'  'magic crystal'
    "The crystal glows with a pure but eerie light. "
    brightness = 3
;
```

Once the player character has this magic crystal and is carrying it around with him or her, it'll provide light to see by wherever he or she goes.

For a slightly more sophisticated light source we can use the LightSource class. When a LightSource is lit it will be listed as "(providing light)" (via the ThingState mechanism we've just described above). Whether or not a LightSource is lit is determined by whether its isLit property is true or nil. We can use the makeLit(lit) method to switch between these two states. The brightness of a LightSource changes between the values of its brightnessOn and brightnessOff properties depending on whether or not it's lit; LightSource defines its brightness property as:

```
brightness { return isLit ? brightnessOn : brightnessOff; }
```

By default, brightnessOn is 3 and brightnessOff is 0. If we wanted to implement a LightSource with a faintly glowing LED that made it just visible even when off, but a lower powered light even when lit, we could, for example, change these values to 2 and 1.

A very common kind of light source both in IF games and in real life is a flashlight (or "torch" in British parlance) which can be switched on and off. TADS 3 provides the Flashlight class for this kind of light source. Flashlight inherits from LightSource and hence has all the same methods and properties, but in addition can be turned on and off by the player (through commands like **switch flashlight on**). A Flashlight also has an isOn property to determine whether or not it is switched on, and a makeOn(stat) method to turn it on and off programmatically; this method also takes care of keeping the isLit property in sync with the isOn property, so if we want to change the on/off or lit/unlit status of a Flashlight in our program code we should always do so using makeOn() (as opposed to manipulating the isOn or isLit properties directly or by using makeLit()). By default a Flashlight starts switched off and unlit; if we want a Flashlight to start switched on and lit we can set the initial value of its isOn property to true (isLit will then automatically follow).

Although, as its name suggest, the Flashlight class can most obviously be used for portable flashlights/torches, it can of course be used for any kind of light source we want the player to be able to switch on and off, including lamps, lanterns, searchlights, and light-switches.

If we want to enforce the condition that the flashlight should only work when it has a battery in it, we have to write our own code to do it. One approach would be to make the flashlight a `RestrictedContainer` that can contain only the battery, and then apply the appropriate checks for the presence and removal of the battery, something along the lines of:

```
flashlight: RestrictedContainer, Flashlight 'flashlight/torch' 'flashlight'
   validContents = [battery]
   makeLit(stat)
   {
       if(stat && !battery.isIn(self))
           failCheck('Nothing happens; presumably because there's no
               battery. ');
       else
          inherited(stat);
   }
   notifyRemove(obj)
   {
       if(obj == battery && isLit)
       {
            makeLit(nil);
            "Removing the battery makes the flashlight go out. ";
       }
   }
   notifyInsert(obj, newCont)
   {
       inherited(obj, newCont);
       if(obj == battery && isOn)
       {
            makeLit(true);
            "The flashlight comes on as you insert the battery. ";
        }
   }
;
```

In this case we allow `isLit` to become decoupled from `isOn`, since removing the battery (say) will stop the flashlight from being lit, but it won't move the on-off switch; and if the switch is left in the 'on' position then presumably the flashlight should light as soon as the battery is re-inserted.

In this example, we assume a battery with an effectively infinite life. If we wanted to implement a battery with a finite life, we would have to make our code more sophisticated still. Alternatively, we could consider using the `FueledLightSource` class (another subclass of `LightSource`), which might more typically be used for things like oil lamps (which burn a certain amount of fuel each turn and go out when the fuel is exhausted).

The most important new property `FueledLightSource` defines is `fuelLevel`, which, as its name suggests, defines the current amount of fuel in the `FueledLightSource`. We can set this to an initial value, and increase it as appropriate if, for example, the player pours oil into the lamp. The library assumes that a fueled light source consumes one unit of fuel for each turn it's lit, and there's probably no good reason ever to change that, unless perhaps we have a number of different fueled light sources all using the same kind of fuel (oil, say), but consuming it at different rates. If we do want to change the rate of consumption, the easiest place to do it may be in the **consumeFuel()** method; e.g., to double the rate of consumption we could define:

```
  consumeFuel(amount) { fuelLevel -= 2 * amount; }
```

The need to do this is, however, likely to be rare.

By default the library assumes that a **FueledLightSource** is its own fuel source. If we want to change that assumption we can override the **fuelSource** property to point to some other object (a battery, say), in which case this other object must define the methods **getFuelLevel()** and **consumeFuel(amount)**. Both of these would probably work via a **fuelLevel** property, e.g.:

```
battery: Thing 'small red battery*batteries'  'small red battery'
   fuelLevel = 40
   getFuelLevel() { return fuelLevel; }
   consumeFuel(amount)  { fuelLevel -= amount; }
;
```

Incidentally, this battery is twice as generous with its initial fuel level as the library default for a **FueledLightSource**, which defines an initial fuelLevel of 20 (which we can, of course, easily change if we like by overriding it).

FueledLightSource defines two further methods that may be of particular interest, **sayBurnedOut()** and **burnDaemon()** (neither of which take any arguments). The first of these simply displays a message to say that the fueled light source has burnt out, so if we don't like the default message provided by the library we can provide our own simply by overriding this method. The **burnDaemon()** method is responsible for consuming the fuel, and making the light go out once the fuel is exhausted, calling the **sayBurnedOut()** method at that point. Although we probably won't often need to change that behaviour, we might often want to add to it in order to provide the player with warning messages when the light is about to go out, something like:

```
lamp: FueledOilLamp 'oil lamp/lantern*lamps' 'oil lamp'
   burnDaemon()
   {
      switch(fuelLevel)
      {
         case 3: "The lamp grows dim. "; break;
         case 2: "The lamp's flame starts to gutter. "; break;
         case 1: "The lamp seems about to go out. "; break;
      }
      inherited;
   }
;
```

A particular sub-type of **FueledLightSource** that the library implements as standard is the **Candle**. This is a type of fueled light source than can be lit by setting fire to it. It defines the properties **okayBurnMsg** and **outOfFuelMsg** which we can, if we wish, override to provide our own descriptions of the Candle being lit or going out. It also defines the method **canLightWith(obj)** which by default simply returns true; this doesn't mean that a Candle can be lit with every single object in the game, it just means that by default a Candle doesn't put any extra restrictions on what it can be lit with (beyond the obvious fact that it must be something that's itself alight). The **canLightWith(obj)** method thus allows us to put further restrictions on what will light the Candle.

Although the class is called Candle, it can be used for any kind of fueled light source that provides light by burning, and so could be used for (flaming) torches, oil lanterns, coal fires and the like. One point to note, however, is that when a Candle burns out, it is simply reported as being out of fuel. This may be fine for an oil lantern, coal fire, or some kind of torches, but when a real-life candle burns out what we're left with is generally some kind of stub, so we might want to provide some mechanism for transforming the candle into a stub. One way might be to write a **desc** method that changes the description of the candle according to its **fuelLevel**, and a **sayBurnedOut()** method that transforms it into a stub; for example:

redCandle: Candle 'red candle*candles' 'red candle'

```
    desc()
    {
        if(fuelLevel > 1)
           "It's about <<fuelLevel>> centimetres long<< isLit ? ', and burning
             merrily' : ''>>. ";
        else
           "It's just a stub. ";
    }

    sayBurnedOut()
    {
       "The candle goes out, leaving a mere stub. ";
       name = 'candle stub';
       initializeVocabWith(name);
    }

    bulk = (fuelLevel / 4)
    weight = (bulk)
    outOfFuelMsg = 'There\'s only a stub left; it's impossible to light it. '
;
```

One thing we can light a candle with is another lit candle. To add that behaviour to a Candle, we just need to include **FireSource** in its class list (before **Candle**, since **FireSource** is a mix-in class). All the **FireSource** class does is to allow **burn with** to pass the verify() stage; the rest is left up to the object being burned. One of the commonest kinds of fire sources used to set light to other objects are matchsticks, and the library provides a **Matchstick** class to represent those. A TADS 3 **Matchstick** is self-igniting (the player can light a matchstick without the help of another object), burns for a short time (by default two turns) with a feeble light (by default a brightness of 2) and, while it's still burning, can be used to light burnable objects such as Candles. Once it's burned out, a **Matchstick** simply disappears.

We can change the length of time a Matchstick burns for and its brightness when lit by overriding its **burnLength** and **brightnessOn** properties.

Matchstick is one of the three TADS 3 classes that has an associated pair of ThingStates:

```
matchStateLit: ThingState 'lit'
    stateTokens = ['lit']
;
matchStateUnlit: ThingState
    stateTokens = ['unlit']
;
```

For further details of how TADS 3 Matchsticks work, look up Matchstick in the *Library Reference Manual*.

Matchsticks don't generally come singly but in books (or boxes) of matches. For this purpose TADS 3 defines the **Matchbook** class, which we might typically use like this:

```
modify Matchstick
    vocabWords = 'match/matchstick*matches*matchsticks'
    name = 'match'
;

Matchbook 'matchbook*matches' 'matchbook'
;

+ Matchstick;
+ Matchstick;
+ Matchstick;
+ Matchstick;
+ Matchstick;
```

This only gives the Matchbook five matches; in practice we may often want to give it rather more, but the principle should be clear from the example.

Exercise 17: Write a short game in which the player character has to explore a small network of dark caves to find a magic glowing crystal. Implement the full variety of different kinds of light sources for exploring the caves. The player character starts only with a book of matches. There's a candle in the first cave (but only with a limited life). Another cave contains an oil lamp, but the supply of oil is separate. Another cave contains a flashlight, and the final cave contains a rusty old box containing the crystal. When you've done, compare your version with the LightFire.t sample game.

11 Nested Rooms

11.1 Types and Characteristics of NestedRoom

Back in chapter 5 we discussed the containment model in TADS 3, and saw how various classes such as `Container` and `Surface` can be used to put things in and on. But while these classes can contain *things*, they can't contain actors, and, in particular they can't contain the player character. If we want the player character or some other actor to be inside or on top of some object, we need to use one of the `NestedRoom` class.

There are four main kinds of `NestedRoom`, and three more specialized kinds we''ll look at later. In this section we'll concentrate on the four common kinds:

- `Chair –` a NestedRoom an actor would typically sit on (but, by default, can also stand on); we'd normally use this for chairs, sofas and the like.
- `Bed` – a NestedRoom an actor would typically lie on (but, by default, can also sit or stand on); we'd normally use this for beds, cots, hammocks and the like.
- `Platform` – a NestedRoom an actor would typically stand on (but, by default, can also sit or lie on); we'd normally use this for things like a stage; we could also use it for rugs, carpets and so forth if we want actors to be able to stand on them, or for the upper surface of desks, tables and the like if we want actors to be able to stand on them.
- `Booth` – a NestedRoom an actor would typically stand *in* (but, by default, can also sit in and lie in); we'd typically use it for large wardrobes, closets, chests and boxes large enough to get in, shallow pits and the like. A `Booth` can be made openable by preceding it with the `Openable` class in its class list.

A NestedRoom is any object that isn't a room but which can contain an actor. All NestedRoom classes have quite a few things in common. For one thing, travel in and out of a NestedRoom is handled a bit differently from travel between Rooms. No travel notifications are triggered, and normally no automatic look around; an actor who enters a NestedRoom is not considered to be leaving the room in which that NestedRoom is located (if I sit on a chair I'm still in the room that contains the chair); likewise leaving a NestedRoom is not counted as entering the room in which it's located (when I get out of the chair I'm not entering the study, I was there all along). Normally the command **out** will take the player character out of whatever NestedRoom s/he's in, but otherwise issuing a movement command (like **east** or **up**) when in a NestedRoom will behave just as if it had been issued when in the enclosing Room (taking the player character out of the NestedRoom via an implicit action before carrying out the travel command).

For the most part, we can use NestedRooms in a fairly straightforward, intuitive way.

For example, to set up a small bedroom with a single bed and a wooden chair, we could just do this:

```
bedroom: Room 'Bedroom'
   "This bedroom is so small that there's little space for anything apart
    from the single bed crammed hard against the wall. The only way out is
    via the door to the east. "
   east = bedroomDoor
   out asExit(east)
;

+ bedroomDoor: Door 'door*doors' 'door'
;

+ Bed, Heavy 'single bed*beds*furniture' 'single bed'
;

+ woodenChair: Chair 'small wooden chair*chairs furniture' 'wooden chair'
  initSpecialDesc = "A small wooden chair next to the bed takes up most
  of the spare space in the room. "
;
```

With this definition, the player could sit or stand on the chair, or sit, stand or lie on the bed. The command **get on bed** will result in the player character lying on the bed, while **get on chair** will result in the player character sitting on the chair. The bed is too heavy to pick up, but the player character can pick up the chair and can also put it on the bed. It's also possible to get on the chair while the chair is on the bed. There is one subtlety: if the player types **out** while the player character is on the bed, the player character will get off the bed; if the player then types **out** again the player character will go through the door. If, however the player types **east** while the player character is on the bed, the player character will get off the bed (via an implicit action) and then go through the door.

11.2 Nested Rooms and Postures

In the previous section we talked about actors sitting on chairs, lying on beds, and standing on platforms. This introduces a concept we haven't formally met before, that of an actor's *posture*. The library defines a `Posture` class and three objects of that class: `standing`, `sitting` and `lying`. By default all actors (including the player character) start out standing. An actor's current posture is held in its `posture` property (which must normally be one of `standing`, `sitting`, or `lying`, unless we define additional postures in our game). We can change an actor's posture with the `makePosture(newPosture)` method, which simply changes the value of the `posture` property.

To create a new posture (such as kneeling), we'd first need to define the actions that put an actor in that posture, (such as KneelAction for the intransitive command **kneel** and KneelOnAction for transitive commands such as **kneel on carpet**). Defining the kneeling posture is then straightforward, even though we need to define quite a few properties and methods:

```
kneeling: Posture
   tryMakingPosture(loc) { return tryImplicitAction(KneelOn, loc); }
   setActorToPosture(actor, loc)  { nestedActorAction(actor, KneelOn, loc); }
   msgVerbIPresent = 'kneel{s} down'
   msgVerbIPast = 'knelt down'
   msgVerbTPresent = 'kneels{s}'
   msgVerbTPast = 'knelt'
   participle = 'kneeling'
;
```

This example serves to show both how to define a new posture, and how the existing postures work. If our game takes place entirely in the present tense (as most games generally do), there's obviously no need to define the `msgVerbXPast` properties. The difference between the first two methods is that `tryMakingPosture()` tries to put the actor into the posture via an implicit action (e.g. "(first kneeling on the carpet)") while the `setActorToPosture()` method does the same thing with an ordinary action (e.g. "Bob kneels on the carpet").

For the most part we probably won't often need to define new postures, and won't have to worry much about the inner workings of the postures the library defines. It is, however, worth being aware of the `participle` property. Suppose we write a room description like this:

```
EdgeOfForest: OutdoorRoom 'Edge of Forest' 'the edge of the forest'
    "You are standing just at the start of a narrow path that snakes off
     mysteriously into the dark forest. "
    north = forestPath
;
```

The trouble is that by mentioning the player character's posture in the room description, we have just given a hostage to fortune; the player who issues the command **sit** or **lie** at this point will immediately turn our room description into a lie. We can avoid this by using the `participle` property of the player character's current posture instead:

```
EdgeOfForest: OutdoorRoom 'Edge of Forest' 'the edge of the forest'
    "You are <<me.posture.participle>> just at the start of a narrow path
     that snakes off mysteriously into the dark forest. "
    north = forestPath
;
```

Our room description will then automatically adapt to whatever posture the player character adopts.

Nested Rooms descending from the `BasicChair` class (as all four Nested Room types introduced in the previous section) define a number of properties and methods relating to actor posture. Unfortunately it's not immediately obvious what these do and how they relate to one another. We'll start with the three posture-related properties:

- **`allowedPostures`** – this is the easiest of the three to understand; this property contains a list of the properties than an actor is allowed to adopt in or on this Nested Room object. For example, if we were using a `Chair` object to represent a large sofa, it may also be possible to lie on the sofa, so we'd override its `allowedPostures` to `[sitting, standing, lying]`. Conversely, if we were using a Chair object to represent the back seat of a cramped vehicle, it might only be possible to sit on it (and not to stand on it), so we might override its `allowedPostures` property to just `[sitting]`. It's a bad idea, however, to override the `allowedPostures` property so as to exclude the most obvious posture for a class (sitting for a `Chair`, standing for a `Platform`, or lying for a `Bed`), since this will effectively break it.
- `obviousPostures` – this also contains a list of postures, but in this case these are the *obvious* postures for the object (e.g. `[sitting]` for a `Chair` or `[lying]` for a `Bed`). The main effect of this is that adopting an allowed posture will not be allowed as an implicit action unless it's also an obvious posture.
- `defaultPosture` – this is the least intuitive of the three posture properties. It does define what one might expect, namely the most obvious posture for each kind of object (standing for a `Platform`, sitting for a `Chair`, or lying for a `Bed`), but it doesn't entirely *work* as one might expect. In particular it does *not* define the posture an actor adopts when entering the object from the outside (so changing the `defaultPosture` of a Bed to sitting will *not* make the player character sit on the bed in response to a **get on bed** command). Instead it controls only the posture an actor adopts when entering this object from a Nested Room contained within this object; for example, if there's a chair on the bed, then the `defaultPosture` property of the bed defines the posture an actor adopts on leaving the chair. It also controls the posture an actor must be in before entering a Nested Room inside this object (e.g. if a chair is on a bed it enforces the curious condition that the player character must be lying on the bed before getting on the chair).

The counter-intuitive behaviour of defaultPosture raises the question what does control what posture an actor adopts on entering a NestedRoom?

In the simplest case the posture may be defined by the player's command: if the player types **lie on bed** or **sit on bed** or **lie on bed** then the player will simply adopt the posture commanded.

The more complex case is where a player types a command like **get on bed** or **board bed**. In such a case the behaviour is governed by the way dobjFor(Board) is defined on the class:

- `Platform` – `dobjFor(Board) asDobjFor(StandOn)`
- `Booth` – `dobjFor(Board) asDobjFor(StandOn)`

- `Chair – dobjFor(Board) asDobjFor(SitOn)`
- `Bed – dobjFor(Board) asDobjFor(LieOn)`

A further complication is that an actor may be moved to a Nested Room via an implicit action (e.g. if the chair is on the bed and the player types **get on chair** the player character will be moved to the bed by an implicit action before sitting on the chair). The implicit action to be carried in this case is defined by the Nested Room's `tryMovingIntoNested()` method; this is variously defined on the relevant classes as:

- `Platform – { return tryImplicitAction(StandOn, self); }`
- `Booth – { return tryImplicitAction(Board, self); }`
- `Bed – { return tryImplicitAction(LieOn, self); }`
- `Chair – { return tryImplicitAction(SitOn, self); }`

There's also a corresponding `tryRemovingFromNested()` method that attempts to remove the actor from Nested Room via an implicit action, but that's more straightforward; it's simply useful to be aware of its existence. The similar `removeFromNested()` takes the current actor (`gActor`) out of the Nested Room via an ordinary (non-implicit) action.

One oddity about the library behaviour (though this may have been altered in 3.0.16) is that the posture adopted in `tryMovingIntoNested()` must match the `defaultPosture` of the object, or else an attempt to move directly to a NestedRoom within the NestedRoom will fail without explanation. For example, suppose we override the bed's `defaultPosture` to `sitting`; if the player puts a chair on the bed and then issues the command **get on chair** we'll see "(first lying on the bed)" followed by nothing; the player character will be left lying on the bed. We'd get similar behaviour if we overrode the bed's `tryMovingIntoNested()` method to make the actor try sitting on the bed (while leaving the `defaultPosture` as `lying`). If we changed both together, all would be well again.

The reason for this is a little arcane: the `NestedRoom.checkMovingActorInto()` first tries to move the actor into the Nested Room and then regards it as a failure if the actor is not in that NestedRoom's `defaultPosture`, simply quitting the action without explanation. Another way we can make things go wrong is by excluding `defaultPosture` from the list of `obviousPostures`. If, for some strange reason, we overrode `obviousPostures` on the bed to just `[sitting]`, then we could get a transcript like this:

>put chair on bed
(first taking the chair)
Done.

>get on chair

You must lie on the bed first.

The reason being that the game won't allow an actor to adopt a posture in/on a Nested Room via an implicit action unless the posture is one of the **obviousPostures** for that NestedRooms.

The moral to be drawn is that NestedRooms are straightforward enough to use if we just accept their behaviour straight out of the box, but can become full of traps for the unwary once we start to customize which postures they're meant to work with. Adding or removing the odd **allowedPosture** is reasonably straightforward; beyond that we need to be very careful. The alternative moral to draw is that when picking which type of NestedRoom to use for any particular object, our first consideration should be what posture we want the player character to adopt in response to a command to **get on** the object in question. If, for example, we want **get on foo** to make the player stand on foo, then we're almost certainly better off making foo a Platform even it it otherwise looks and behaves like a chair or a bed.

11.3 Nested Rooms in Complex Containers

It may happen that we want to make a bed the player character can both lie on and put things under, or a desk s/he can both stand on and put things under and behind. Making a desk the player can stand on is simple enough: we just make it a **Platform** instead of a **Surface**. This automatically allows us to put things on it as well as stand on it. But if we also want to allow objects to be put under and behind the desk, we need to make the desk a **ComplexContainer**. As of version 3.0.17, we can simply do this:

```
desk: ComplexContainer 'sturdy desk*desks'  'sturdy desk'
    "It's large, and quite sturdy enough to stand on. "
    subUnderside: ComplexComponent, Underside { }
    subRear: ComplexComponent, RearContainer { }
    subSurface: ComplexComponent, Platform { }
;
```

Our ComplexContainer desk will then automatically handle commands such as **get on desk**, **stand on desk**, **sit on desk** and **get off desk**, redirecting them from the ComplexContainer object to the subSurface. We can also use a ComplexContainer for something the player can get inside, such as a large wardrobe:

```
+ ComplexContainer, Heavy 'large wardrobe' 'large wardrobe'
    subContainer: ComplexComponent, Openable, Booth { }
    subSurface: ComplexComponent, Surface { }
;

++ ContainerDoor '(wardrobe) door' 'wardrobe door'
;
```

This will work, although with one slight oddity: the commands **stand on wardrobe**,

sit on wardrobe, and **lie on wardrobe**, will cause the player character to stand, sit or lie *in* the wardrobe, not on it.

You may be wondering how the ComplexContainer knows which of its subcomponents to direct **get in** and related commands to. The answer is that it uses the method `getNestedRoomDest(action)` to determine which of the ComplexContainer's subXXXX properties has a NestedRoom attached to it, looking first at the subSurface and then at the subContainer. Most of the time this is fine, and we can just leave ComplexContainer to do its job. Occasionally, however, we may want to implement something like a large crate the player can either get inside on get on, in which case we may need to override `getNestedRoomDest()` to decide which subcomponent to use. For example:

```
+ ComplexContainer 'large crate' 'large crate'
    subContainer: ComplexComponent, Booth { }
    subSurface: ComplexComponent, Platform { }
    getNestedRoomDest(action)
    {
        if(action.ofKind(EnterAction) ||
            gAction.getEnteredVerbPhrase.find(' in'))
            return subContainer;

        return subSurface;
    }
;
```

The tricky point here is that **get in crate** and **get on crate** both trigger the action **board crate**; **stand in crate** and **stand on crate** both trigger the StandOn action; and so on for sit in/on and lie in/on, whereas players will obviously expect these commands to work differently depending whether their command used **in** or **on** (or **into** or **onto**). The above example deals with this by checking whether the player's command includes a word beginning with 'in'; if it does, or if an **enter** command was used, we select the subContainer; otherwise we select the subSurface. For most purposes this should work well enough.

Note that we used the method `getEnteredVerbPhrase` (called on gAction) to get at the form of the command the player entered. This returns a string in the form 'get in (dobj)' or 'put (dobj) on (iobj)' (in other words using dobj and iobj as placeholders for the actual objects, thus focusing on the *form* of the command rather than the specific objects it refers to). This string will always be in lower case, and since it is always in this standard form it is quite easy to use it to determine which actual verb and preposition the player actually typed. Here we use it to see if the preposition 'in' or 'into' was used; we can test for both at once simply by testing for the presence of the string ' in', which can only occur if the player type a command like **get into crate** or **stand in crate**.

Note too that we can use a Bed, Platform, Chair or Booth as a ComplexComponent (Platform and Booth are likely to be the most useful), and this will allow the player character or another actor to get on or in a ComplexContainer, but the library provides no mechanism to allow an actor to get under or behind something. If you require such functionality in your game, your best bet is probably to download the ConSpace extension from the IF-Archive.

11.4 Staging Locations

An actor has to be in one of a Nested Room's staging locations before s/he can enter that NestedRoom (for example, the player character needs to be on the bed before entering the chair when the chair was on the bed). By default the `stagingLocations` property of a NestedRoom is simply `[location]`; we need to be in a NestedRoom's location to enter that NestedRoom. As we've just seen above this is something we can override if we need to, and since `stagingLocations` is a list property, we can define multiple valid staging locations for any given Nested Room.

Since there may be more than one valid staging location, NestedRoom defines the `chooseStagingLocation()` method to select which of them is to be used. This method simply chooses the actor's current location if that's one of the staging locations, or otherwise returns the `defaultStagingLocation()`.

In turn, the `defaultStagingLocation()` method simply searches through the list of `stagingLocations` until it finds one which is known to be a staging location for this NestedRoom and then returns that (this means that the order in which staging locations are listed in the `stagingLocations` property is significant, since the first one in the list is likely to be used in the default).

Whether or not any given staging location *loc* is known to be a staging location for this NestedRoom is determined by calling the `isStagingLocationKnown(loc)` method, which simply returns true by default. Note that this only effects whether *loc* can be used in an implicit action. For example, if we changed this method on a chair always to return nil, and then put the chair on the bed, the command **get on chair** would get the response "You can't do that from here." (instead of an implicit action to take the player character onto the bed first). On the other hand, **get on chair** would still work when the player character was already in the chair's staging location (by default, simply its location).

But there's another potential pitfall here: if we simply overrode chair's `isStagingLocationKnown(loc)` method to return nil, we'd find that although we could get on the chair, we wouldn't be able to get off it again. The reason is that when an actor leaves a Nested Room s/he's taken to the location defined in that Nested Room's `exitDestination` property, and that by default `exitDestination` is defined to take on the value of `defaultStagingLocation()`. But if we've defined the chair as having no known staging location, `defaultStagingLocation()` will return nil, leaving the player character nowhere to go when leaving the chair. One way round that, if we really must make the chair's staging locations unknown, is to override its `exitDestination` to `(location)`.

Using one NestedRoom as the staging location for another is, in the main, reasonably straightforward, and apart from the ComplexContainer case noted above, or the case of some subtle puzzle we're trying to devise, we can normally just use the standard library behaviour without worrying about it. If, however, we want to use a Nested Room as a staging location for a TravelConnector, things can rapidly become more complicated. An example might be where the player character has to stand on a chair in order to reach a window high up in the wall, in order to climb out through the window. Part of the problem is that TravelConnector presents a different interface for staging locations; on a TravelConnector we have to define the **`connectorStagingLocation`** property (not the **`stagingLocations`** property), and that can only hold a single object, not a list. But that's only the beginning of the potential difficulties. To find out more about the problems of using NestedRooms as staging locations for TravelConnectors, along with some suggested solutions, see the article on "Using Nested Rooms as Staging Locations" in the *TADS 3 Technical Manual*.

11.5 Other Features of Nested Rooms

11.5.1 Nested Rooms and Bulk

It's worth bearing in mind that actors have bulk and Nested Rooms have a **`bulkCapacity`**; in other words, who or what can fit into a NestedRoom may be limited by bulk.

By default, the bulk of a Person (the class used for humans NPCs) is 10; the **`bulkCapacity`** of a Chair is 10, and the **`maxSingleBulk`** of virtually all objects is 10 by default. That means that a default TADS 3 chair is only just large enough to hold a default TADS 3 Person; not even a small object of bulk 1 will fit on the chair alongside the Person, and even one such small object on the chair will be sufficient to prevent a Person from sitting there.

Of course we can easily override the **`bulkCapacity`** of our chairs to some larger value; the point is to remember to do so if we need to. For example, a sofa designed to seat three people would need a **`bulkCapacity`** of at least 30 (and perhaps a bit more to allow room for a few small objects on the sofa besides). Again, if we decide to increase the default bulk of a Person (because we want more gradations in bulk between the tiniest objects and a human being), we need to remember not only to increase the **`bulkCapacity`** of Chair but also the **`maxSingleBulk`** of NestedRoom accordingly.

11.5.2 Dropping Things in Nested Rooms

If the player character drops an object while in a Nested Room the game must decide whether the thing that has just been dropped ends up in the Nested Room or in the Nested Room's location. Dropping something while on a Platform is likely to result in the object falling onto the Platform. Dropping something while on a Chair is likely to result in the object falling into the chair's enclosing room. This is what the library enforces in both cases.

Put more formally, `BasicChair` (from which all the kinds of Nested Room we've been looking at descend) overrides the `getDropDestination(obj, path)` method (where *obj* is the object being dropped; for most objects we don't need to worry about *path*), so that it returns the drop destination of the enclosing location, if there is one, and self otherwise. The method is overridden again on `BasicPlatform` (from which `Platform` and `Booth` descend) simply to return self (so that something dropped on a Platform will fall onto the Platform). If we want to change this behaviour, this is the method we need to override. For example, on a tiny rug we might define:

```
tinyRug: Platform 'tiny rug*rugs' 'tiny rug'
    getDropDestination(obj, path) { return inherited BasicChair(obj, path); }
;
```

Conversely on a very large sofa we might define:

```
sofa: Chair, Heavy 'huge sofa/settee*sofas' 'huge sofa'
    getDropDestination(obj, path) { return self; }
;
```

Then anything dropped while the player character is on the tiny rug will fall to the drop destination of the rug's location, while anything dropped while the player character is on the huge sofa will fall onto the sofa.

11.5.3 Enclosed Nested Rooms

By default a Nested Room is open to its immediate surroundings; an actor sitting on a chair in the lounge is still treated as being in the lounge; a **look** command will describe the lounge, and everything in the lounge is reachable from the chair. If, however, the player character goes inside an openable Booth and closes it, then unless the Booth is made of some transparent material (`glass`, `fineMesh`, or `coarseMesh`), s/he will not be able to see out into the enclosing room. In such a case the Nested Room's `roomName` property will be used as the name of the player character's current location, and the `roomDesc` property used to provide an internal description of the NestedRoom for the purposes of a **look** command. Of course, if the player character shuts himself inside an opaque NestedRoom with no source of light, he won't be able to see anything at all; but we can then override the `roomDarkName` and `roomDarkDesc` properties of the NestedRoom to provide custom versions just as we can for a DarkRoom. By default, the `roomName` of a `NestedRoom` is simply its name.

11.6 Special Kinds of Nested Room

There are three more kinds of Nested Room, used for more specialized purpose: **NominalPlatform**, **Vehicle**, and **HighNestedRoom**.

The purpose of **NominalPlatform** is to provide somewhere to put an NPC we want described as “standing in the doorway” or “leaning against a lamp-post”. In most cases it’s probably easier to obtain the same effect by playing with the NPC’s **specialDesc** (or his ActorState’s **specialDesc**), so we won’t go any further into this class here; interested readers can look up NominalPlatform in the *Library Reference Manual*.

The **Vehicle** class is rather more interesting. A **Vehicle** is a kind of Nested Room that moves around in response to movement commands while the player character is inside it. We might use it for, say, a bicycle, or a pedal-car, or a go-kart. We’d typically mix it in with some other kind of NestedRoom class, such as Chair (if it’s the sort of vehicle we’d sit on, like a bicycle) or perhaps Booth (if it’s the sort of vehicle we’d get inside). For example, we might define a simple bicycle as:

```
bike: Chair, Vehicle 'bicycle/bike*bikes bicycles'  'bicycle'
   allowedPostures =  [sitting]
;
```

Note that when an actor is moving around in or on a vehicle, it's the vehicle rather than the actor that passed as the *traveler* parameter to the various methods that take a traveler parameter. Thus, for example, the **VehicleBarrier** class is defined as:

```
class VehicleBarrier: TravelBarrier
    canTravelerPass(traveler) { return !traveler.ofKind(Vehicle); }

    /* explain why we can't pass */
    explainTravelBarrier(traveler)
    {
        reportFailure(&cannotGoThatWayInVehicleMsg, traveler);
    }
;
```

If we wanted, we could define our own VehicleBarrier objects to allow some kind of vehicles through but not others; e.g.

```
modify VehicleBarrier
    allowedVehicles = []
    canTravelerPass(traveler)
    {
        if(allowedVehicles.indexOf(traveler))
           return true;

       return inherited(traveler);
    }
;

allowBikeBarrier: VehicleBarrier
    allowedVehicle = [bike, motorBike]
;
```

The **allowBikeBarrier** would allow an actor through on foot, or when riding the bike or the motor-bike, but not when in any other vehicle (see section 4.5 above, if you need reminding what a **TravelBarrier** is).

The final special kind of NestedRoom is the **HighNestedRoom**. This is a special kind of NestedRoom that's notionally too high up to get in or out of (a high shelf or bunk bed, perhaps). This should normally be mixed in with another NestedRoom class (**Bed** or **Platform**, say), with **HighNestedRoom** coming first. HighNestedRoom simply provides customized messages to say that it's too high up to get in or out of (in its **cannotMoveActorToStagingLocation()** and **cannotMoveActorOutOf()** methods) and defines **stagingLocations** as an empty list, so that there's no way in or out. Presumably we wouldn't want a HighNestedRoom to be permanently inaccessible (it might then just as well be a **Distant**), so we'd need to find some way of changing the its **stagingLocations**; either by making its **stagingLocations** property a method that returns non-empty list under certain circumstances, or by dynamically adding and removing items to and from its stagingLocations elsewhere in our code. For example, it order to make a high platform that's reachable only via a ladder and only when that ladder is in its leaning against platform state we might do something like this:

```
+ HighNestedRoom, Platform, Fixture 'high shelf*shelves' 'high shelf'
    stagingLocations = (ladder.leaningAgainst == self ? [ladder] : [] )
;
```

Here, we're assuming that **leaningAgainst** is a custom property of the ladder object that we've implemented appropriately elsewhere in our code.

11.7 Nested Rooms and OutOfReach

Although the **OutOfReach** class is logically quite distinct from any Nested Room class, in practice the two may often work together. As its name at least partly suggest, **OutOfReach** is a mix-in class that can put a container, and optionally it contents, out of reach except under author-defined conditions. **OutOfReach** defines the following methods to this end:

- **cannotReachFromOutsideMsg(dest)** – the message used to say that an actor can't reach into this object from the outside to touch *dest* (if we override this method it should return a single-quoted string).
- **cannotReachFromInsideMsg(dest)** – the message used to say that an actor can't reach out of this object from the inside (if we override this method it should return a single-quoted string).
- **canObjReachContents(obj)** – returns true if the object *obj* (normally an actor) can reach the contents of the OutOfReach object from the outside. By default this just returns nil.

- **canObjReachSelf(obj)** – returns true if the object *obj* (normally an actor) can reach the OutOfReach object itself from the outside. By default this returns the value of canObjReachContents(obj).
- **canReachFromInside(obj, dest)** – returns true if the object *dest* outside the OutOfReach object can be reached by the object *obj* (typically an actor) while *obj* is inside the OutOfReach object. By default this simply returns nil.
- **canReachSelfFromInside(obj, dest)** – returns true if an *obj* (typically an actor) can reach the OutOfReach object from inside it. By default this returns the value of canReachFromInside(obj, self).
- **tryImplicitRemoveObstructor(sense, obj)** – by default this returns nil to mean that we can't do anything about an object's being out of reach. In some cases we could override this to attempt an implicit action (e.g. getting out of the OutOfReach object to reach the object that's outside it).

From this set of method definitions, three things should be apparent:

1. An **OutOfReach** won't do anything terribly interesting unless we override at least some of these methods.
2. The **OutOfReach** class must be mixed in with some kind of container class.
3. Some of the **OutOfReach** methods (those concerned with try to reach outside from within) are only meaningful if **OutOfReach** is mixed in with a **NestedRoom** class (otherwise, there could be no actor inside).

We saw how a **HighNestedRoom** could be used to implement a high shelf that can only be reached via a ladder; but that was a high shelf the player could actually climb onto via the ladder. A more common kind of high shelf might be one that's too small to climb on to, but supports some objects that the player character wants to get hold of. This is one way in which we'd use an **OutOfReach** in conjunction with a **NestedRoom**, since we'd probably arrange for there to be some kind of **NestedRoom** (a chair say) that the player character could stand on in order to reach the high shelf:

```
+ OutOfReach, Surface, Fixture 'high shelf*shelves'  'high shelf'
   canObjReachContents(obj)
   {
      return obj.isIn(woodenChair);
   }
;

++ torch: Flashlight 'torch/flashlight*torches' 'flashlight'
;

+ woodenChair: Chair 'wooden chair' 'wooden chair'
;
```

In this example, there's a flashlight (which the player presumably needs to visit some dark room or other) resting on the high shelf. In order to reach either the high shelf or anything on it (such as the flashlight) the player character needs to stand on the wooden chair.

As mentioned in passing above, if an actor is in a NestedRoom, then unless it's a closed Booth, it's assumed that the actor can reach anything in the room. This may sometimes seem a bit unrealistic, especially if the room is much bigger than a broom closet. If I'm sitting on a chair in the study, I'm unlikely to be able to take a book off the bookcase unless my chair just happens to be right next to the right section of shelf. If I'm lying on my bed in a reasonably large bedroom I'm unlikely to be able to reach something that's lying on the dressing table set against the far wall.

Combining OutOfReach with a NestedRoom allows us to model this situation. At a first approximation we might write:

```
+ bed: OutOfReach, Bed, Heavy 'bed*beds'  'bed'
    canReachSelfFromInside(obj) { return true; }
    canObjReachContents(obj) { return true; }
;
```

In this case we don't want to prevent anyone from outside the bed touching either the bed or its contents, and someone on the bed can obviously reach the bed itself, so we override two of the methods accordingly. The result is that the bed now behaves much like an ordinary TADS 3 Bed, except that an actor on the bed can't reach anything outside the bed. If we wanted to be more discriminating we could override **`canReachFromInside(obj, dest)`** to allow access to a restricted range of objects.

A further sophistication would be to automate the process of getting off the bed if the player tries to take something outside the bed from inside the bed. Rather than being told "You hat is too far away" it would make for smoother game player to remove the player from the bed (with "(first getting off the bed")) and then allow the player to take the hat. We can achieve that by overriding **`tryImplicitRemoveObstructor()`** to attempt an implicit action to get the actor off the bed:

```
+ bed: OutOfReach, Bed, Heavy 'bed*beds'  'bed'
    canReachSelfFromInside(obj) { return true; }
    canObjReachContents(obj) { return true; }
    tryImplicitRemoveObstructor(sense, obj)
    {
        return tryRemovingFromNested();
    }
;
```

For further details of OutOfReach (though we've covered most of them here) look up OutOfReach in the *Library Reference Manual*. For more details of NestedRooms (and there is more to see), look up NestedRoom and its subclasses (as well as the BasicLocation superclass) in the *LibraryReferenceManual*.

Exercise 18: Create a one-room game consisting of the Player Character's bedsit. This will contain a bed (of course) under which is a drawer containing a pillow. There’s also a desk, a swivel chair (too unstable to stand on) and an armchair (too heavy to move), as well as a large sofa that’s large enough to lie on. Above the desk is a high bookshelf on which is a solitary book. To reach the shelf or the book the player character must stand on the desk. On another wall is a high bunk bed which can only be reached via a ladder that’s currently stored under the desk. Beneath the bunk bed is a wooden bench seat, attached to the wall; there’s insufficient headroom under the bunk bed to stand on the bench, and insufficient headroom above it to stand on the bunk. Also on the floor under the bunk is a sleeping cat. Finally, there’s an openable wardrobe that’s large enough to walk into; inside the wardrobe is a hanging rail on which is a solitary coat-hanger. If you’re feeling really adventurous make it so that in general an actor inside a NestedRoom can’t reach outside it (there will need to be exceptions to this), and will automatically be taken out of the NestedRoom if s/he tries. Once you’ve got as far as you want to with this, compare your version with the bedsit.t sample game.

12 Locks and Other Gadgets

12.1 Locks and Keys

We've come across several things than can be open and closed: doors, some containers, and some booths. When we're writing IF we often want to make such things lockable too.

12.1.1 Lockable

The simplest kind of lock in TADS is provided by the `Lockable` class. To use this class we simply add Lockable to the front of the class list, so to make a lockable door we'd use a class list of `Lockable, Door;` a lockable booth (such as a lockable wardrobe) would use `Lockable, Openable, Booth`; and a lockable container would be `Lockable, OpenableContainer`, except that the library already provides a `LockableContainer` class that combines these classes. Since `Lockable` is a mix-in class, it's generally a good idea to put it first in the class list. More significantly, we don't need a key to lock or unlock something that's of the `Lockable` class; the kind of lock envisaged is a bolt or paddle; all that's needed to lock or unlock a `Lockable` is to issue a **lock** or **unlock** command. `Lockable` is therefore probably of limited use, since making something `Lockable` doesn't provide any real obstacle. The `Lockable` class is probably most useful for implementing one side of a door, either something like a bathroom door that can only be locked or unlocked from the inside, or something like a front door that needs a key to lock and unlock from the outside but which can be locked or unlocked with a bolt or paddle on the inside.

12.1.2 KeyedLockable

The more common kind of lock in Interactive Fiction is one that requires a key. For this TADS 3 provided the `KeyedLockable` class. Like Lockable, this is a mix-in class that should come first in any class list, hence `KeyedLockable, Door` or `KeyedLockable, Openable, Booth`. TADS 3 provides a `KeyedContainer` class that already combined `KeyedLockable, Openable, Container`. Note that when defining a pair of Doors (i.e. the two game objects that represent the two sides of the same physical door) it's perfectly legal to make one side a `Lockable, Door` and the other side a `KeyedLockable, Door.`

A keyed lockable obviously needs a key, and to make something a key we simply define it to be of the `Key` class. To define which keys unlock which locks, we list the keys that work with a particular KeyedLockable in that KeyedLockable's `keyList` property. So, for example, if the front door can be locked and unlocked with the brass key, we'd define:

```
+ frontDoor: KeyedLockable, Door 'front door*doors'  'front door'
    keyList = [brassKey]
;
```

Since **keyList** is a list property, it can contain more than one key. So, for example, if there was also a skeleton key in the game that opened a wide variety of doors we could define:

```
+ frontDoor: KeyedLockable, Door 'front door*doors'  'front door'
    keyList = [brassKey, skeletonKey]
;
```

It's thus perfectly possible to make the same key work on a number of different locks, and for a number of different keys to work on the same lock.

Once the player character has successfully used a key on a lock once, the game remembers that the player character knows that this key works on this lock. Thereafter (unless we override this behaviour) the process of unlocking and locking that particular lock will be automated through implicit actions, provided the player character still has the appropriate key. For example, if the front door is locked, but the player character already knows that the brass key unlocks it, then issuing a command to go through the door will cause the door to be implicitly unlocked and opened, like this:

```
>n
(first unlocking the front door with the brass key, then opening it)
```

The game keeps track of which keys are *known* to fit which locks in the lockable objects **knownKeyList** property. So, if there are keys the player character should start the game already knowing about, these can be defined on the appropriate lock's **knownKeyList**. For example, the player character presumably starts the game knowing which key unlocks his or her own front door, so if it's the player character's own front door we're implementing we could do something like this:

```
+ frontDoor: KeyedLockable, Door 'front door*doors'  'front door'
    keyList = [brassKey, skeletonKey]
    knownKeyList = [brassKey]
;
```

This is perhaps as much as we need to know to make locks and keys work reasonably well in our game, but there are other properties and methods we can tweak to customize their behaviour. The following are all defined on **Lockable**, and are also applicable to **LockableWithKey** (where any of them behave differently on **LockableWithKey**, this will be noted below).

- **autoLockOnOpen** – if this is true, then an **open** command will trigger an implicit **unlock** command if we're locked. By default this is set to the value of `lockStatusObvious` (for which, see below).

- `initiallyLocked` – if this is true (the default) then this object starts the game locked. Note that if we're defining a pair of linked objects (such as the two sides of a door) this should be defined on the master object (the one that doesn't use -> in its template).
- `lockedDesc` – an adjective describing this item as either locked or unlocked; the library defaults are simply 'locked' and 'unlocked'; if the property is overridden it should be with a method that returns one single-quoted string when the item is locked and another when it's unlocked.
- `lockStatusObvious` – this should be true for an item whose locked/unlocked status can be seen from visual inspection, e.g. because the lock is operated by a bolt or paddle the position of which can be seen. The default value is true.
- `lockStatusReportable` – although this is similar to `lockStatusObvious`, it is used for a slightly different purpose. By default the library appends "It's locked" or "It's unlocked" to the description of any Lockable object. If we find this aesthetically displeasing we can set this property to nil to suppress such reports. The library default is for this property to be true unless the item is open, in which case reporting that it's unlocked seems redundant. Note that both `lockStatusObvious` and `lockStatusReportable` have to be true for the "It's locked" messages to appear.
- `isLocked()` - use this method to test whether or not the item is locked (it returns true if it is locked and nil otherwise)
- `makeLocked(stat)` – use this method to lock or unlock the item under program control; if *stat* is true the item is locked, otherwise it's unlocked.

`LockableWithKey` additionally defines (or overrides) the following:

- `keyList –` as we've already seen, this contains the list of keys that can lock or unlock this item.
- `knownKeyList` – as we've also already seem, this contains the list of keys that the player character knows can lock or unlock this item.
- `lockStatusObvious` – this has the same meaning as on `Lockable`, but on `LockableWithKey` the default value is nil, since we can't normally tell just by visual inspection whether a keyed lock is currently locked or unlocked.
- `rememberKnownKeys` – if this is true (the default), then each time the player successfully uses a key to lock or unlock this item, that key will be added to the item's `knownKeyList`.
- `isKeyKnown(key)` – returns true if the player character knows that *key* fits this lock. By default this method returns true if *key* is in the `knownKeyList`.

- `keyFitsLock(key)` – returns true if *key* fits this lock. By default this returns true if *key* is in the keyList, but it's conceivable that we might occasionally want to override this to apply some other criterion.
- `keyIsPlausible(key)` – returns true if *key* looks as if it might fit this lock. For example, if the lock is a Yale lock, we know that only a Yale key could plausibly fit it. Similarly, if it's a lock operated by a card key, then only a card key could plausibly lock or unlock this item. Similarly, some keys might be obviously too large or too small for some locks; a large iron key that might unlock a church door is never going to open a small tin box any more than the tiny silver key that might unlock the small tin box could ever unlock the church door. By default this routine simply returns true, but we could override it if we wished to be more discriminating. The point of the routine is to help the parser decide which key the player means in cases of ambiguity, so that, for example, if the method has been properly set up and the player types **unlock small tin box with key** the parser will select the small silver key rather than the large iron key without bothering with player with a disambiguation question, since it's obvious which key the player must mean.

12.1.3 Keyring

As noted above, anything we want to be usable as a key should be of the `Key` class. If we like, we can also define or or more objects of the `Keyring` class. A `Keyring`, as its name suggests, is a special kind of object for holding keys. If an actor is holding a `Keyring` when s/he takes a `Key`, that `Key` will automatically be put on the `Keyring`. Keys on a Keyring can be used to lock and unlock things without removing them from the Keyring, and if an actor in possession of a keyring executes an **unlock** command without specifying what key to use, we will automatically test each key on the ring to find the one that works.

It may be that a keyring that's suitable for conventional keys wouldn't be the natural place to put card-keys or the like. By default, a `Keyring` will take anything that's a Key, but if there are different kinds of Keys in our game, and we don't want a particular Keyring to take all of them, we can override that Keyring's i`sMyKey(key)` method to return true only for those keys we want that Keyring to hold.

12.1.4 IndirectLockable

In addition to items that can be locked and unlocked with a simple paddle mechanism or with some sort of key are doors and containers that use some other mechanism, such as a safe door that uses a combination lock, or a door operated by a button or lever. For such situations the library defines the class `IndirectLockable`. This is used in the same way as `Lockable`, and like `Lockable` it is a mix-in class that should generally appear first in the class list of any object definition. When defining the two sides of a door it is again perfectly legal to make one side `IndirectLockable` and the other either `Lockable` or `LockableWithKey`. The distinguishing feature of an `IndirectLockable` is that the **lock** and **unlock** commands won't work on it, since the player has to find some other means (such as operating the combination lock, pressing a button or pulling a lever) to lock or unlock the item.

`LockableWithKey` inherits from Lockable and therefore inherits all the same methods and properties. In addition it defines or overrides:

- `cannotLockMsg` – a message explaining that the player cannot **lock** this item directly. To override this (which we'll frequently want to do) define this property as a single-quoted string.
- `cannotUnlockMsg` – a message explaining that the player cannot **unlock** this item directly. To override this (which we'll also frequently want to do) define this property as a single-quoted string.
- `lockStatusObvious` – this means the same as on `Lockable`, but the default is nil.

12.2 Control Gadgets

As we've just seen, if we define something (typically a door or container) to be an `IndirectLockable` we need to provide some kind of external mechanism to lock or unlock it. For this purpose we might use one of the control gadget classes provided by the library, which include `Button`, `Lever`, `OnOffSwitch`, `Settable` and `Dial`. Of course we can also use these classes to control any other kind of contraption we like.

12.2.1 Buttons, Levers and Switches

The simplest of these classes is probably `Button`. By default a Button simply goes *click* when it's pushed; to make it do anything more interesting we need to override its `actionDobjPush()` method, for example:

```
study: Room 'Study'
    "There seems to be some sort of door in the oak-panelling on the north
     wall. Next to this is large brown button. "
    north = panelDoor
;

+ panelDoor: IndirectLockable, Door 'door*doors' 'door'
   cannotLockMsg = 'Maybe that's what's the brown button is for. '
   cannotUnlockMsg = (cannotLockMsg)
;

+ Button, Fixture 'large brown button*buttons'  'brown button'
  dobjFor(Push)
  {
     action()
     {
       "A loud <i>click</i> comes from the door in the panelling. ";
       panelDoor.makeLocked(!panelDoor.isLocked);
     }
  }
;
```

A `Lever` is slightly more complicated in that it has two states, pulled or pushed (determined by the value of its `isPulled` property). When `isPulled` is true the player has to **push** the lever to move it; when `isPulled` is nil the player must **pull** the lever to move it. The player can also simply **move** the lever to toggle between one state and the other. Any of these actions uses the `makePulled(pulled)` method to switch states, and this is probably the most convenient method to override to make the lever actually do anything. For example, suppose instead of an indirectly lockable door in the wood panelling, we have a hidden door that only becomes apparent when it is operated by a concealed lever. We could do this with:

```
study: Room 'Study'
    "Oak panelling covers the walls. A matching oak desk stands in the middle
     of the room. "
    north = panelDoor
;

+ panelDoor: HiddenDoor 'secret panel door*doors' 'secret door'
;


+ desk: ComplexContainer, Heavy 'oak wooden desk*desks*furniture' 'oak desk'
   subSurface: ComplexComponent, Surface { }
   subUnderside: ComplexComponent, Underside
   {
      dobjFor(LookUnder)
      {
         action()
         {
            if(!hiddenLever.discovered)
              "You find a small concealed lever fixed to the underside of the
               desk. ";
            inherited;
         }
      }
   }
;


++ hiddenLever: Lever, Component 'hidden concealed small lever' 'small lever'
   subLocation = &subUnderside
   makePulled(pulled)
   {
      inherited(pulled);
      panelDoor.makeOpen(pulled);
      if(pulled)
        "A secret door slides open in the north wall. ";
      else
        "The secret door in the panelling slides shut. ";
   }
;
```

A variation on the `Lever` is the `SpringLever`, which is a spring-lever that returns to its starting position when pulled (making it functionally equivalent to a `Button`). Although `SpringLever` inherits `isPulled` and `makePulled()` from Lever, neither is relevant to its operation, and to make a SpringLever do anything useful we need to override is `actionDobjPull()` method. For example, to implement the previous lever under the desk as a spring lever we might do this:

```
++ hiddenLever: SpringLever, Component 'hidden concealed small lever*levers'
    'small lever'
   subLocation = &subUnderside
   dobjFor(Pull)
   {
      action()
      {
        panelDoor.makeOpen(!panelDoor.isOpen);
        if(panelDoor.isOpen)
          "A secret door slides open in the north wall. ";
        else
          "The secret door in the panelling slides shut. ";
      }
   }
;
```

A slightly different kind of control is the `OnOffControl`, which, as its name suggests, is a control the player can **turn on** or **turn off** (or **switch** on and off). An OnOffControl has an `isOn` property that determines whether it's on or off, and an `onDesc` property which returns 'on' or 'off' appropriately. Switching an `OnOffControl` between its on and off states is handled in its `makeOn(val)` method, which is probably the most convenient method to override to make the `OnOffControl` do something interesting.

For example, suppose we wanted to implement a light switch that controls a light bulb in some other part of the room. We could define:

+ OnOffControl, Fixture 'light switch*switches' 'light switch'

```
    makeOn(val)
    {
       inherited(val);
       lightBulb.makeLit(val);
       "The light bulb <<val ? 'comes on' : 'goes out'>>. ";
    }
;
```

A `Switch` is identical to an `OnOffControl` except that the player can additionally use the command **flip** and **switch** to toggle a Switch between its on and off states (i.e. isOn being true or nil). A `Flashlight`, which we have already met, is a combination of the `Switch` and `LightSource` classes, suitably linked so that its `isOn` and `isLit` properties remain in sync.

12.2.2 Controls With Multiple Settings

The various types of control gadgets we have met so far have at most two states, but there are various kinds of control (e.g. the slider on a thermostat or a dial on a radio) that can have multiple settings. The ancestor class for all such multiple-setting classes in TADS 3 is the `Settable` class, which is a possible class to use for slider-like controls.

The principal properties and methods of the `Settable` class are:

- `curSetting` – the item's current setting, which can be any (single-quoted) string value. This is updated as the item is set to a different setting.
- `canonicalizeSetting(val)` – by default this just returns *val*. It could be used, for example, to convert *val* to lower case before testing whether or not it's a valid setting for this item, thereby making commands for setting this item not case-sensitive.
- `isValidSetting(val)` – by default this also just returns true, but we'd normally need to test it to ensure that any proposed new setting was indeed valid. Note that when this is called *val* has already been converted to canonical form by `canonicalizeSetting()`.
- `makeSetting(val)` – this is the method that changes the setting, by changing `curSetting` to *val*. Note that when this called *val* has already been converted to canonical form by `canonicalizeSetting()`. This is probably the most convenient method to override if we want changing the setting of this item to have any interesting effect.

Two further properties of possible interest are `okaySetToMsg` and `setToInvalidMsg` which return the (single-quoted) strings to be used either to acknowledge the change of setting or to complain that the proposed new setting is invalid.

In most cases the settings we can set a `Settable` to will either be a range of numbers or a finite set of strings. In that case the simplest thing to do if we're implementing a Settable that isn't a dial (it's a slider, say), is to "borrow" the `canonicalizeSetting()` and `isValidSetting()` methods from either the `NumberedDial` or `LabeledDial` class (which we'll meet shortly below). Or we might simply *use* one of these classes for our slider or whatever without worrying that the player can also use commands like **turn slider to 10** or **turn slider to amber** to set the slider. To "borrow" these methods we could define a numbered slider like this:

```
+ Settable, Component  'slider*sliders'  'slider'
    "It can be set to any number between <<minSetting>> and <<maxSetting>>.
     It's currently set to <<curSetting>>. "
    minSetting = 0
    maxSetting = 70
    curSetting = '60'
    canonicalizeSetting(val) { return delegated NumberedDial(val); }
    isValidSetting(val) { return delegated NumberedDial(val); }
    makeSetting(val)
    {
       inherited(val);
       val = toInteger(val);
       if(val < 40)
         "Gosh it's becoming cold in here! ";
       if(val > 70)
         "It's starting to become rather too warm! ";
    }
;
```

Likewise for a labeled slider we could define:

```
+ Settable, Component  'slider*sliders'  'slider'
   "The slider can be set to any of
   <<stringLister.showSimpleList(validSettings)>>. It's current set to
   <<curSetting>>. "
   validSettings = ['red', 'yellow', 'blue' ]
   canonicalizeSetting(val) { return delegated LabeledDial(val); }
   isValidSetting(val) { return delegated LabeledDial(val); }
   makeSetting(val)
   {
      inherited(val);
      if(val == 'red')
        "A klaxon starts to sound. ";
   }
;
```

This should become a bit clearer when we look at NumberedDial and LabeledDial below, and in any case it might be easier just to use `NumberedDial` or `LabeledDial` and override `verifyDobjTurnTo()` if we want to disallow the **turn to** command on our slider.

Note that by default a `Settable` can be set only with commands like **set slider to whatever**. If our Settable is a slider, the player might reasonably try to **move slider to 10** or **push slider to green**. The easiest way to cater for that is simply to modify the grammar of the SetTo command:

```
modify VerbRule(SetTo)
  ('set' | 'slide' | 'move' | 'push' | 'pull') singleDobj 'to' singleLiteral
    :
;
```

A specializiation of `Settable` is the `Dial` class, which simply allows **turn dial to x** as well as **set dial to x**. It’s unlikely that we’ll ever want to use the raw Dial class in a game; we’re much more likely to use one of its more fully-features subclasses, `NumberedDial` or `LabeledDial`.

A `NumberedDial` is a dial that can be set to any one of a range of integer settings. The range of settings is specified by the `minSetting` and `maxSetting` properties. The one tricky thing to look out for is that the `minSetting` and `maxSetting` properties must be specified as *numbers* while the `curSetting` property is a (single-quoted) *string*.

A typical use for a NumberedDial might be as a combination lock. For example:

```
++ NumberedDial, Component 'black dial*dials'  'black dial'
    "The dial can be turned to any number from 0 to 99; it's currently at
     <<curSetting>>. "
   minSetting = 0
   maxSetting = 99
   combination = [21, 34, 45]
   storedSettings = []
   makeSetting(val)
   {
      inherited(val);
      storedSettings += toInteger(val);
      if(storedSettings.length > 3)
        storedSettings = storedSettings.sublist(storedSettings.length - 2);

      if(storedSettings == combination)
      {
         location.makeLocked(nil);
         "As you turn the dial to <<val>>, a quiet click comes from
           <<location.theName>>. ";
      }
      else
         location.makeLocked(true);
   }
;
```

This assumes, of course, that this **NumberedDial** is attached to something (like a safe or strongbox) that can be locked or unlocked.

A **LabeledDial** can be turned to any one of a number of predefined settings. The allowable settings are listed in its **validSettings** property. However we define these strings, and whatever the player types, both the setting proposed by the player and the settings in the **validSettings** property are converted to upper case before they're compared, so that a **LabeledDial** is not case-sensitive.

A **LabeledDial** might thus typically be defined along the following lines:

```
+ LabeledDial, Component  'red dial*dials'  'red dial'
    "The dial can be turned to any of the settings
    <<stringLister.showSimpleList(validSettings)>>. It's currently set to
    <<curSetting>>. "
  validSettings = ['OFF', 'SAFE', 'OVERDRIVE', 'REVERSE']
  makeSetting(val)
  {
     inherited(val);
     switch(val)
     {
        case 'OFF':
          "The widget-mangler falls silent. "; break;
        case 'OVERDRIVE':
          "The widget-mangler starts vibrating alarmingly. ";
        /* other interesting stuff */
     }
  }
;
```

Note the use of `stringLister.showSimpleList(validSettings)` in the description. This will take the contents of the validSettings property and display it as properly formatted list, like "OFF, SAFE, OVERDRIVE and REVERSE". We could just write out this list by hand, of course, and in a dial with few settings it's probably simpler to do so; but if our dial has many settings using the stringLister to list them not only saves us the bother of typing them twice, but it ensures that the description does in fact correspond to the `validSettings` property, which both makes it more typo-proof and makes it easier to keep the two in sync if we start adding to and subtracting from the list of `validSettings`.

Exercise 19: Try implementing the following game. The player character is outside the home of a blackmailer. Knowing him to be out, the player wants to burgle his house to recover an incriminating letter. The player character carries a small black case holding a skeleton key and a key-ring, and also has a small flashlight (if you want to be really sophisticated you can try to see whether the player refers to it as a 'flashlight' or a 'torch' and then use American or British English from then on accordingly). The front door can be unlocked either with the skeleton key or with the key hidden under a nearby flowerpot. Once inside the hall the player character must disable the burglar alarm before going any further into the house. The alarm is controlled by a numeric keypad inside a box by the front door. The correct combination is the date (year) that was written over the outside of the front door. To unlock the box containing the keypad requires either the skeleton key or a small silver key that falls to the ground when the player character pulls a peg on the nearby hat-stand. Once the alarm has been turned off, the player character can go into the study. On one wall of the study is a panel than needs to be opened to gain access to the safe. In the study is a desk on which is a small wooden box. On the side of the box is a slider, which can be set to the names of four different composers; the box is unlocked when the slider is used to spell out the word OPEN from the initial letters of these composers. Inside the box is a key that can be unused to unlock the drawer of the desk. This contains a notebook in which is written the cryptic message "Advertising is safe" together with the combination of the safe. There's a TV in the study which can be turned on and off with a switch, and changed to different channels with a dial. Turning it on and switching it to the advertising channel will open the panel in the wall. The player character can then go through the open panel into a small cubby-hole containing the safe. From inside the cubby-hole the panel can be opened and closed by means of a lever. The safe has dial which must be turned to each of the numbers in the combination for the safe to be unlocked. Once the safe is unlocked it can be opened and the letter retrieved. The game is won when the player character walks away from the house carrying the letter.

Once you've got as far as you want to implementing or thinking about how to implement the game, take a look at the sample game LockGadget.t.

13 More About Actions

13.1 Message Properties

As we've seen, it's possible to use macros like `illogical('You can't do that. ')` in verify routines, or `reportFailure('You see no reason to strip off here. ')` in check routines to stop an action and display a message explaining why. We can also use macros like `mainReport('You pick up the vase. ')` and `defaultReport('Taken. ')` to report the carrying out of actions. If, however, we look at the way such actions are defined in the library, we won't see them defined with string arguments like this; instead we'll see them used with property pointers, and then sometimes with additional arguments, like `illogical(&notASurfaceMsg)`, or `reportFailure(&actorCannotSeeMsg, gIobj, self)`, or `defaultReport(&okayTakeMsg)`.

That property pointers are used here is a clue that what is happening: the messages displayed by these macros are generated by the methods and properties of some object. Where there's more than one argument, the additional arguments are arguments passed to this method.

In action processing the message object used for this purpose is defined in `gActor.getActionMessageObj()`. Thus, for example, when we see a macro like

```
illogical(&notASurfaceMsg);
```

Then the message that this produces is the single-quoted string returned from:

```
gActor.getActionMessageObj().notASurfaceMsg;
```

When the current actor is the player character, then unless we change the library's default behaviour `gActor.getActionMessageObj()` will be the object `playerActionMessages`, defined in the file msg_neu.t. This means that the message displayed, for example, by

```
reportFailure(&actorCannotSeeMsg, gIobj, self);
```

Will be the single-quoted string returned from:

```
playerActionMessages.actorCannotSeeMsg(gIobj, self);
```

The main reason the library does it this way is to isolate the language-dependent parts of the library into a couple of files. If someone wants to translate the library into another language, all they need to do is to translate the messages in these two files. By using message properties rather than actual strings throughout, the main part of the library remains language-independent.

When we're writing a game in a particular language we don't generally need to worry about this too much; we can simply continue to use single-quoted strings in our `illogical()`, `reportFailure()` and `mainReport()` macros, and all will be well. On the other hand, it's useful to know how the library messages work if we want to change them.

If we want to change a library message globally, then one way to do it is to modify the `playerActionMessages` object. For example, to change the default message for putting something on something that's not a `Surface` we could do this:

```
modify playerActionMessages
    notASurfaceMsg = '{You/he} can\'t put anything on {the iobj/him}. '
;
```

If we want to change several default library messages in our game, this is perfectly good way to go about it; we could either read through the msg_neu.t file noting the messages we wanted to override, or search through it to hunt for particular message we want to change when they come up in the course of testing our game.

When we want to change the messages used on individual objects or custom classes we need a different approach. Of course we could always do this:

```
+ Enterable ->frontDoor 'large house*houses'  'large house'
   "It's a large, three-storey house. "
   iobjFor(PutOn)
   {
      verify() { illogical('You can't reach the roof from here! '); }
   }
;
```

But this becomes a little cumbersome when we want to override several messages on several objects. After all we're not changing the logic here, just the message that's displayed, and it seems a little like overkill to have to rewrite the entire verify() method just to change the message it displays.

In fact, we don't need to. In order to allow for this kind of situation the library first looks at the objects involved in a command to see whether any of them define the property in question before going to the `gActor.getMessageObj()` property. So we could, almost, get the effect we wanted simply by defining:

```
+ Enterable ->frontDoor 'large three-storey house*houses'  'large house'
   "It's a large, three-storey house. "
   notASurfaceMsg = 'You can't reach the roof from here! ' // not quite right
;
```

The slight problem with this is that the library looks for the appropriate message property on all the objects involved in an action, so our custom message would also be used if the large house were the direct object of a **put on** command. This could result in our getting a transcript like this:

>put large house on small tray
You can’t reach the roof from here!

In this context our custom message looks quite inappropriate (although we might not actually get it in this particular case, since the action would presumably fail when trying to implicitly take the house, but the principle remains the same). What we need is a means to tell the library to use our custom message only when the large house is the indirect object of a **put on** action, and not the direct object. We can do that by using the `iobjMsg()` macro. We can similarly use the `dobjMsg()` macro to indicate that we only want our custom message to be used when the large house is the direct object of the command:

```
+ Enterable ->frontDoor 'large three-storey house*houses'  'large house'
   "It's a large, three-storey house. "
   notASurfaceMsg = iobjMsg('You can't reach the roof from here! ')
   cannotCleanMsg = dobjMsg('You hardly have time to clean the entire house! ')
   cannotEatMsg = 'You appetite isn't that big! '
;
```

This ensures that we don’t get inappropriate responses to **put house on tray** or **clean boots with house**.

Where an action can only have one object (as with **eat house**) this isn’t an issue, so we don’t need to use the `dobjMsg()` macro. Where a message can relate to an action that takes two objects (for example there’s both a Clean action and a CleanWith action, and both use `cannotCleanMsg`) it’s important always to use the `dobjMsg()` or `iobjMsg()` to ensure that we only get our custom message under the appropriate circumstances.

There’s actually three different kinds of value you can assign to a message property on an object to be used instead of the same message property on `playerActionMessages`:

- You can define it to be a single-quoted string (or a property containing a single-quoted string), as in the examples above. Then that single-quoted string will be used in place of whatever the message property on `playerActionMessages` would have been.
- You can define it to be another message property; then that message property will be used on `playerActionMessages` instead of the one you are overriding.
- You can define it to be nil, in which case the original message property of `playerActionMessages` will be used. There’s no point in making your custom message property unconditionally nil (you may as well not define it at all), but it can be a method that returns nil under certain circumstances; the `dobjMsg()` and `iobjMsg()` use this technique to ensure that the `playerActionMessages` property is used if they’re not called on the direct or indirect object respectively.

For more details of how this works, look up the **`MessageResult`** class in the Library Reference Manual and read the comments in the code for the **`resolveMessageText()`** method. From this you may note one further point; some properties of **`playerActionMessages`** are simple properties taking no arguments; some are methods with one or more arguments. When defining your message override on an object (or class) you can either define it without any arguments (even if it's defined with arguments on **`playerActionMessages`**) or with the same arguments as the method has on **`playerActionMessages`**. For example, **`playerActionMessages`** defines a **`cannotTasteMsg(obj)`** method. If you want to define your own **`cannotTasteMsg`** on a particular object you can either do it as a simple property:

```
cannotTasteMsg = 'You don\'t want to put {that dobj/him} in your mouth! '
```

Or as a method with the same arguments as the method on playerActionMessages:

```
cannotTasteMsg(obj)
{
   gMessageParams(obj);
   return '{You/he} really do{es}n\'t want to try tasting {that obj/him}! ';
}
```

There's no need to use the second form unless you actually want to use the parameters in your method.

Defining message properties in this way saves having to override verify, check and action methods just to change a message, but the catch is we have to know which message property to override. It's possible to figure it out from the *Library Reference Manual*, but all too often, by the time we've done that we might as well have overridden a method! To find the appropriate message property it's quicker and easier to look it up in the quick reference chart that's available from tads.org, at http://www.tads.org/howto/ActionMessages.zip. The same quick-reference chart is also provided as an appendix to *Getting Started in TADS 3*.

For more information on library message properties, read the "Library Messages" section at the end of the article on "Action Results" in the *TADS 3 Technical Manual*. You may also find it helpful to read the preceding section ("Action") in the same article. For a parallel account of the material just covered here, you could read the "Messages" section in Chapter 4 of *Getting Started in TADS 3*.

13.2 Stopping Actions

We have already encountered the **`exit`** macro, which can be used to stop an action in its tracks (typically in a check() routine, but as we shall see, it can be used elsewhere as well). We should now take a slightly closer look at this and other ways of stopping actions before they're allowed to run their normal course.

The effect of `exit` is to halt the action just at the point where the `exit` statement appears, and skip straight to the end of turn processing (other actors' actions, if any are pending, Fuses and Daemons). Note, however, that the `exit` statement skips only the rest of the command processing for the current action on the current object. So for example, if the player enters the command **take the apple, the carrot and the banana** and an `exit` macro is encountered in the course of taking the carrot, taking the carrot will be skipped but the game will go on to try to take the banana. Likewise, if the player enters several commands on the command line, only the current action is skipped by an `exit` macro. For example, if the player had entered **take the carrot then go north** and an `exit` macro prevented the taking of the carrot, the player character would still try to go north.

It's debatable whether we'd actually want the subsequent commands on the command line to go ahead once one has failed. If immediately to the north of the player character's current location is a demon donkey who can only be appeased with a carrot, then allowing **go north** to succeed after **take carrot** fails may have disastrous consequences for the player character which the player was deliberately trying to avoid. TADS 3 therefore allows an option to cancel the entire command line once an action fails; to do this, set the `cancelCmdLineOnFailure` property of `gameMain` to true (for more details look up this property on `GameMainDef` in the *Library Reference Manual*).

Similar to the `exit` macro is the `exitAction` macro. The main difference is that `exitAction` skips over the rest of the action handling and skips to the afterAction phase (which we'll see more about below), while `exit` skips over the afterAction stage as well.

Neither of these macros skips over the iteration of multiple objects (the **take apple, banana and carrot** case); if we wish to cancel iterating the same command over the remainder of the objects, we need to call `gAction.cancelIteration()`.

As we've just seen, we can set `gameMain.cancelCmdLineOnFailure` to true in order to cancel the remainder of a command line once one action fails, but this is a global action. If we wanted this cancellation to occur only in the case of the carrot and the demon donkey, say, then we'd need some way of leaving the global option at its default while cancelling the remainder of command line if taking the carrot fails. We can do that by throwing a `CancelCommandLineException` e.g.:

```
carrot: Food 'carrot*carrots vegetables' 'carrot'
    "It's a particularly fine specimen, as carrots go. "
    dobjFor(Take)
    {
       check()
       {
          if(hilda.canSee(self))
          {
            reportFailure('Hilda snatches the carrot out of your hand and
             returns it to the vegetable rack. ');
             throw new CancelCommandLineException;
            }
        }
    }
;
```

This introduces a feature of TADS 3 programming we haven't encountered before; we'd better explain it now.

13.3 Coding Excursus 17 – Exceptions and Error Handling

We've just used some code that throws something called an **Exception**. In fact, that's what the **exit** and **exitAction** macros do too. These are, after all, macros, which means they're really a convenient abbreviation for some other code. Their full definitions are:

```
#define exit throw new ExitSignal()
#define exitAction throw new ExitActionSignal()
```

To explain these seemingly mysterious definitions, we need to explain a little about how TADS 3 handles *Exceptions*.

A TADS 3 Exception may be some kind of error condition, or it may just be used (as in the examples above) as a convenient means of breaking out of a procedure prematurely and skipping to some later point. In general, then, an Exception represents some kind of unusual situation. More specifically, an Exception is an object of the **Exception** class (or, more likely, of one its subclasses), that encapsulates some kind of information about the exceptional situation.

The Exception mechanism has two main parts: throwing and catching. We have already seen examples of an Exception being thrown, namely via a **throw** statement. To throw an Exception of the **MyException** class we simply use a statement like:

```
throw new MyException;
```

As with any other dynamically created class, an Exception can have a constructor (defined in its **construct()** method) to which parameters can be passed when the Exception is created; this could be used to store additional information about the exceptional circumstances that resulted in the Exception being thrown.

Once an Exception has been thrown, program execution jumps to the next enclosing **catch** statement relevant to that kind of exception. For this to work, the Exception must have been thrown in a block of code protected by a **try** statement. The general coding pattern is:

```
someRoutine()
   try()
   {
      doSomeStuff();
   }
   catch(MyException myexc)
   {
     /* do something with myexc */
   }

   catch(SomeOtherException oexc)
   {
     /* do something with oexc */
   }

   finally
   {
     /* clean up afterwards */
   }
;
```

The code in the `try` block can be as extensive as we like. Here we envisage it calling a `doSomeStuff()` method. If an Exception is thrown in the `doSomeStuff()` method, or a method called by the `doSomeStuff()` method (and so on to any depth of nesting), execution will still jump to the `catch` section of `someRoutine()` (unless `someRoutine()` defines its own `try...catch` block to handle Exceptions). There can be any number of catch blocks; the one that will be used is the first one to match the class of Exception that has been raised (where matching means that the Exception that has been raised is either of the same class as the class listed at the start of the parentheses following the `catch` keyword or is a subclass of that class). Thus, for example, if `doSomeStuff()` threw an Exception of the `MyException` class or of a subclass of `MyException`, it would be caught by the first `catch` statement. The Exception object that has been thrown is then assigned to the local variable named at the end of the parenthesis (we can use any name we like for this). Thus, for example, if `doSomeStuff()` threw a `MyException`, the `MyException` object would be assigned to the local variable `myexc`, so that we could then do something with it if we wished (such as displaying an error message from one of its methods or properties).

The `finally` clause is optional, but must come last if present. The code in the finally block is executed before we leave `someRoutine()` however `someRoutine()` is terminated (whether by `return` or `goto` or any other means). A `finally` block can therefore be used to ensure things get tidied up even if an Exception has been raised.

This explanation is highly compressed; it's more to alert you to the existence of the Exception-handling mechanism than to give a full and detailed explanation. For a fuller explanation, read the chapter on "Exceptions and Error Handling" in Part III of the *TADS 3 System Manual* and look up the Exception class in the *TADS 3 Library Reference Manual*. It's also worth looking at the explanations of `throw` and `try` towards the end of the "Procedural Code" chapter in Part III of the *TADS 3 System Manual*.

13.4 Reacting to Actions

We have seen how the objects involved in an action can respond to it, but it's also possible to make other objects in scope react to it, both before it takes place and after it has taken place. A reaction that occurs before the action takes place can prevent the action happening at all (usually by means of the **exit** macro).

All the objects in scope get a chance to intervene beforehand in their **beforeAction()** method. For example:

```
bob: Person 'bob/man'  'Bob'
    isHim = true
    isProperName = true
    beforeAction()
    {
        inherited;
        if(gActionIs(Yell))
        {
             reportFailure('There\'s no need to shout; Bob can hear you
                perfectly well. ');
             exit;
        }
    }
;
```

If we want the player's *location* to intervene, however, we have to use its **roomBeforeAction()** method; e.g.:

```
lowCave: Room 'Low Cave'
   "There's not much headroom here. "
   roomBeforeAction()
   {
      if(gActionIs(Jump))
        failCheck('You\'d better not jump here; you\'d bump your head on the
          low ceiling. ');
   }
;
```

Note that because Room descends from Thing, calling **failCheck(msg)** is equivalent to calling **reportFailure(msg)** followed by **exit**.

When exactly these two methods are run depends on the value of an option set in **gameMain.beforeRunsBeforeCheck**. If this is true (the current default) then **beforeAction()** and **roomBeforeAction()** run between the verify and check stages of the action. If it is nil, then these before notifiers are run between the check and action stages. This latter is arguably the better option: if an action fails at the check stage it isn't going to be carried out, so that it's not then really appropriate for other objects to react to it.

Objects and locations can also react to actions after the event, through their **afterAction()** and **roomAfterAction()** methods (the latter used on the actor's location, a Room or Nested Room). For example:

```
lowCave: Room 'Low Cave'
  "There is very little headroom here. "
  roomAfterAction()
  {
     if(gActionIs(Jump))
       "Ouch! You bang your head on the ceiling. ";

     if(gActionIs(Yell))
       "Your shout echoes round the cave. ";
  }
;

+ vase: Thing 'vase*vases' 'vase'
  "It looks very delicate. "
  afterAction()
  {
    if(gActionIs(Yell) && !vase.location.ofKind(Actor))
      "The vase vibrates alarmingly. ";
  }
;

+ bob: Person 'bob/man*men' 'Bob'
   isHim = true
   isProperName = true
   afterAction()
   {
      inherited;
      if(gActionIs(Take) && gDobj == vase)
        "<q>Be careful with that!</q> Bob admonishes you. ";
   }
;
```

We call inherited on the `beforeAction()` and `afterAction()` methods of Bob, by the way, since the library already defines something here that we don't want to disable (we'll see more about that in the next chapter).

There are one or two other places we can intervene. Before `roomBeforeAction()` and `beforeAction()` are called (in that order) `beforeAction()` is called on the current action. Just before the check stage the `actorAction()` is called on the actor, and we could use that to restrict the actor's actions when blindfolded or tied up, for example:

```
me: Actor
    actorAction()
    {
        if(isTiedUp && gActionIs(Stand))
          failCheck('You can\'t stand up while you\'re all tied up. ');
    }
    isTiedUp = nil
;
```

Conversely `afterAction()` is called on the action (the `gAction` object) after `roomAfterAction()` has been called on the current location. Once the action has been carried out for every item in the list (which may be only one object for a command like **take vase** but which could be several objects for a command like **take all**) the method `afterActionMain()` is called on the action, and this in turn calls the

afterActionMain() method of every object registered in the current object's **afterActionMainList**. This list can be be constructed by calling **callAfterActionMain(obj)** to add an action to the list; but it is only meaningful to do so while the action is in progress. This last mechanism is primarily intended for use in summarizing the transcript, something we'll briefly return to in Chapter 19.

For a full account of what happens when, see the article on "The Command Execution Cycle" in the *TADS 3 Technical Manual*. This may, however, be an article you will want to leave reading until later.

13.5 Reacting to Travel

Travelling is an action like any other action, and can be reacted to in the same way. However, there's a special set of methods for reacting to travel and it's generally more convenient to use these specialized travel-related methods rather than the more generic beforeAction and afterAction methods.

We've already met one way of reacting to travel, namely by overriding the **noteTraversal(traveler)** on the TravelConnector via which the travel is taking place. In this instance the *traveler* parameter may be the actor whose travelling (if he or she is on foot) or it may be the Vehicle the actor is travelling in. When this method is called, travel is already taking place; the various beforeTravel notifications have already been dealt with, so this method is a good place to carry out the side effects of travel, such as displaying a message describing it. We can also display a message describing travel in the **travelDesc()** method of a TravelMessage or TravelWithMessage (which call **travelDesc()** from **noteTraversal()**).

The travelling equivalent of **beforeAction()** is **beforeTravel(traveler, connector)**. Once again *traveler* may be the actor (if on foot) or the Vehicle in which the actor is travelling; *connector* is the TravelConnector via which the *traveler* is about to travel. If we want to, we can cancel the travel before it gets going by using the **exit** macro in this method, for example:

```
riverBank: OutdoorRoom 'Bank of River' 'the bank of the river'
   "A narrow bridge spans the river to the north. "
    north = bridge
;

+ troll: Person 'mean mean-looking troll/beast*beasts trolls'  'troll'
   "A true mean-looking beast! "
   beforeTravel(traveler, connector)
   {
      if(traveler == me && connector == bridge)
         failCheck('The troll blocks your path with a menacing growl! ');
      inherited(traveler, connector);
   }
;
```

There's a couple of points to note in this example. Firstly, once again, we call the inherited method. It is a good idea to get into the habit of always doing this on Actors (and ActorStates, which we'll meet in the next chapter), since if it's not needed, it'll do no harm, but it frequently is needed and omitting it then is likely to break something (a fairly easy way of introducing bugs into a game).

The second point to note is a second easy way to introduce bugs: suppose we later went back to modify **riverBank** by adding a TravelMessage to its **north** property:

```
riverBank: OutdoorRoom 'Bank of River' 'the bank of the river'
   "A narrow bridge spans the river to the north. "
    north: TravelMessage { ->bridge
      "You walk cautiously onto the bridge. " }
;
```

Now when the player character tries to go north from the river bank, the TravelConnector by which he's attempting to travel is no longer the **bridge** but the anonymous TravelConnector on **riverBank.north**, so that the troll will no longer block the player character's path. One way to avoid this problem is to define the beforeTravel method on the troll as:

```
+ troll: Person 'mean mean-looking troll/beast*trolls beasts' 'troll'
   "A true mean-looking beast! "
   beforeTravel(traveler, connector)
   {
      if(traveler == me && connector == riverBank.north)
         failCheck('The troll blocks your path with a menacing growl! ');
      inherited(traveler, connector);
   }
;
```

This will then work whether **riverBank.north** is left as **bridge** or subsequently changed to a TravelMessage (or some other kind of TravelConnector).

The most convenient travelling equivalent of **roomBeforeAction()** is **leavingRoom(traveler)**, which would need to be overridden on the Room the *traveler* (actor or vehicle) is about to leave. This is in turn called by **travelerLeaving(traveler, dest, connector)** which we could override if for any reason **leavingRoom()** won't do the job for us (e.g. because we need to test where the *traveler* is going); in that case we should need to make sure we also called the inherited method, or we'd probably end up breaking something.

The travel equivalent of **actorAction()** is **actorTravel(traveler, connector)**, where *traveler* and *connector* have the same meanings as before. Once again, if we do override this method, we must be careful to call **inherited(traveler, connector)** somewhere in the overridden method.

Note that all the travel methods we have just seen are called on the room, or objects in the room, that the actor is just about to leave. The next set of methods, equivalent to the afterAction stage, are all called on the room, or objects in the room, that the player has just entered (or is just entering) at the end point of travel.

The travel equivalent of `afterAction()` is `afterTravel(traveler, connector)`. This is called on all the objects in scope in the new location just as the actor is entering it. So, for example, we could have:

```
+ bob: Person 'bob/man*men'  'Bob'
   isHim = true
   isProperName = true
   afterTravel(traveler, connector)
   {
      if(traveler == me)
       "<.p><q>Ah, there you are!</q> Bob greets you. ";
      inherited(traveler, connector);
   }
;
```

In practice we'd probably code this greeting a little differently (as should become apparent in the next chapter), but the example serves to illustrate `afterTravel()` well enough.

The equivalent method to call on the room the actor is just entering is `enteringRoom(traveler)`. One potential catch with it is that this method is called before the room description is displayed; this may not be a problem, depending on what you want to do, but if you want to display something *after* the room description then it is a bit of a problem.

One work around is to use `travelerArriving(traveler, origin, connector, backConnector)` method instead, and display your text after calling the inherited method:

```
marsh: OutdoorRoom  'Marsh'
    travelerArriving(traveler, origin, connector, backConnector)
    {
        inherited(traveler, origin, connector, backConnector);
        if(traveler == me && me.getWeight() > 10)
          "<.p>You feel yourself start to sink into the marsh! ";
    }
;
```

You may also want to use `travelerArriving()` if you need to use any of its arguments, in particular *origin* (the room the *traveler* has just arrived from) or *connector* (the TravelConnector via which the *traveler* has just arrived). But if you do override `travelerArriving()` make sure you call its inherited method somewhere, otherwise you won't get a description of the room the player character has just entered!

13.6 NPC Actions

Actions carried out by NPCs (non-player characters, the other actors in our game besides the player character, who is controlled by the player) present no particular problems in TADS 3, since the library handles them in virtually the same manner as it does actions carried out by the player character. If you actually need an action to work differently for the player character and for NPCs you can test the identity of **gActor**, either with **gActor == gPlayerChar** or with **gActor == me** or with **gActor.isPlayerChar()**; for example:

```
largeBox: OpenableContainer 'large box*boxes'  'large box'
    dobjFor(Take)
    {
       check()
       {
          if(gActor.isPlayerChar())
            failCheck('You\'re too much of a weakling to pick it up; you\'ll
              have to persuade someone else to carry it for you. ');
       }
    }
;
```

The main thing we need to take care of if we want actions to work for actors other than the player character is making sure any messages (or other output text) we write will work as well for other actors as they do for the player character. In practice this means that we must write them all with parameter substitution strings, not just as straightforward text. For example, if an actor other than a player might put down the vase, then instead of writing something like:

```
vase: Container 'priceless cut glass vase*vases'  'priceless vase'
    okayDropMsg = 'You carefully lower the vase to the ground. '
;
```

We must write:

```
vase: Container 'priceless cut glass vase*vases'  'priceless vase'
    okayDropMsg = '{You/he} carefully lower{s} the vase to the ground. '
;
```

Then we'll get "You carefully lower the vase to the ground" or "Aunt Mildred carefully lowers the vase to the ground" as appropriate (if you're a bit uncertain about message parameter strings – {you/he} and the like – read the article on "Message Parameter Substitution" in the *TADS 3 Technical Manual*).

To make an NPC carry out an action we can use the macros **newActorAction()** or **nestedActorAction().** We'd use the first of these to make an NPC carry out a brand new action and the second to make an NPC carry out one action as part of another. Both macros are called with two or more arguments: the first argument is the actor who is to carry out the action; the second is the action to be carried out. Any further arguments are the objects on which the action is to be carried out. So, for example, we could have:

```
newActorAction(bob, Jump);
newActorAction(bob, Take, redBall);
newActorAction(bob, PutIn, redBall, blueBox);
```

The first of these would make Bob jump; the second would make Bob take the red ball; the third would make Bob put the red ball in the blue box.

Note that all of these are ways of making Bob act ‘spontaneously’ (under program control); they are not ways in which the *player* or *player character* gives orders to Bob, they are ways in which the *game author* can make Bob do things.

Making NPCs carry out actions is only a small part of implementing NPC behaviour. In the next chapter we shall go on to see what else we can do with NPCs in TADS 3.

14 Non-Player Characters

14.1 Introduction to NPCs

Non-Player Characters (or NPCs) are any actors (or if you like, any animate objects) that appear in our game besides the Player Character (the character whose actions are controlled by the player). By 'appear' we mean any actor that is actually implemented as an object in the game, and not merely mentioned in a cut-scene, conversation, or some other passing reference.

TADS 3 offers a rich set of tools for controlling NPC behaviour and, in particular, for writing conversations between the player character and NPCs, although making lifelike NPCs remains one of the most difficult and challenging tasks facing any IF author.

The features TADS 3 offers for implementing NPCs are quite fully documented in three linked articles in the *TADS 3 Technical Manual*: "Creating Dynamic Characters", "Choosing a Conversation System" and "Programming Conversations with NPCs"; anyone planning to do any serious work with NPCs in TADS 3 should certainly read all three articles. The present chapter will simply try to give a compressed summary of this material, an overview of what is possible with NPCs in TADS 3, and one or two additional tips along the way.

We'll start with a very brief overview indeed, which we'll flesh out in the following sections. The way TADS 3 is designed means that we generally write very little code on the NPC objects themselves, even for very complex NPCs, since most of an NPC's behaviour is defined on other objects, which we locate (with the + notation) inside the NPC (or Actor) object itself. These other objects include ActorStates, TopicEntries, Conversation Nodes and AgendaItems (all of which we'll look at it in more detail below). An ActorState represents what an Actor is currently doing, generally speaking his or her physical state (such as conversing with the player character, sitting doing some knitting, digging the road, sleeping profoundly, reciting an epic poem, or anything else we care to model in our game). This allows us to define most of the NPC's state-dependent behaviour on the ActorState objects instead of the Actor. TopicEntries represent the NPC's responses to conversational commands (such as **ask bob about susan**, **tell bob about about treasure**, or **show bob the strange coin**). Broadly speaking, each topic is handled by a different TopicEntry; TopicEntries may either be located in the Actor object, or in one of its ActorStates (if they are specific to that state). A Conversation Node is an object representing a particular point in the conversation when certain responses become meaningful (e.g. when the NPC has just asked the Player Character a question to which the replies **yes** or **no** might be appropriate). Finally, an AgendaItem is an object encapsulating something the NPC wants to say or do when the conditions are right and the opportunity arises.

The reason for using all these different kinds of object is that we can thereby avoid a great deal of complicated 'spaghetti' programming with convoluted if-statement and massive switch statements. By distributing the behaviour of a complex NPC over a great many objects of different kinds, we can make each piece of code quite simple, indeed we can often avoid the need to write any code at all, defining much of the NPC's behaviour purely declaratively. This makes for code that is ultimately easier to write, easier to maintain, and less prone to hard-to-track-down bugs.

14.2 Actors

Three kinds of Actor come standard with TADS 3.

First, there is the `Actor` class. The `Actor` class defines most of the behaviour needed for animate objects (NPCs). NPCs defined with the `Actor` class are portable (the player character can pick them up and take them around), so this would be the class to use for small animals such a cats, mice and rabbits.

`UntakeableActor` is a subclass of `Actor`. An `UntakeableActor` is one that can't be picked up and carried around, so we'd typically use it for larger animals like cows, horses and elephants.

`Person` is a subclass of `UntakeableActor.` This is the class we'd normally use for human and human-like NPCs (which might include aliens and intelligent robots). The only real difference between a `Person` and an `UntakeableActor` is that a few of the action response messages (relating to taking and moving) have been customized to be more suitable for a person. In addition, a `Person` is given a default bulk of 10.

There is not a huge difference between these three classes: the player character can give orders to NPCs of all three classes, and we can define conversational responses and agenda items for all three.

The definition of an Actor object can be fairly minimal: we need to specify its `vocabWords` and `name`, and we probably want to give it a description. If it's a person (or gendered animal) we need to remember to indicates its gender by defining `isHim = true` or `isHer = true`. If the Actor has a proper name (e.g. 'Bob' rather than 'the tall man' or whatever) we also need to remember to define `isProperName = true` (so we don't see him referred to as 'the Bob'). So a minimal Actor object definition might look like:

```
mavis: Person 'old frail aunt woman/mavis*women'  'Aunt Mavis'
   "Well past her prime, she is now looking distinctly frail. "
   isHer = true
   isProperName = true
;
```

In practice we'd probably want a bit more than that. In particular, we'd probably want to define some handling for custom actions, or at least customize some of the action response messages, e.g.:

```
mavis: Person 'old frail aunt woman/mavis*women'  'Aunt Mavis' @lounge
   "Well past her prime, she is now looking distinctly frail. "
   isHer = true
   isProperName = true
   cannotKissActorMsg = 'She\'s not of a generation that welcomes outward
      shows of affection. '
   cannotEatMsg = 'Whatever else Aunt Mavis is, she is definitely not that
     tasty. '
   uselessToAttackMsg = 'Beating up Aunt Mavis will not encourage her to be
     generous to you in her will. '
;
```

If the actor's name can change during the course of play (typically because the player character comes to know the actor better), for example changing from 'The tall man' to 'Bob', then it's also very useful to define the `globalParamName` property. This can be defined as any (single-quoted) string value we like, but it's generally a good idea to make it resemble the actor's name. We can then use this string value in a parameter substitution string. For example, if we gave Mavis a `globalParamName` of 'mavis', we could then refer to her in any messages we write as `'{The mavis/she}'`, which would expand to 'the old woman' or 'Aunt Mavis' or whatever her current name property was. This would then enable us to write all our messages about Mavis knowing that they'll use the right name for her whatever the player character knows her as at the point when they're displayed.

To give another example of this:

```
bob: Person 'tall man*men'  'tall man' @highStreet
   "He's a tall man, wearing a smart business suit and sporting a thin
     moustache. "
   isHim = true
   globalParamName = 'bob'
   makeProper(properName)
   {
      name = properName;
      isProperName = true;
      initializeVocabWith(properName.toLower());
      return name;
   }
;
```

Here, `makeProper()` is a custom method we've just defined. It would allow us to write code like `"<q>Hello, I'm <<bob.makeProper('Bob')>>,</q> he introduces himself. "`, which would update his name, `isProperName` and `vocabWords` properties in line with his self-introduction. Descriptions like `"You see {a bob/him} standing in the street"` would then change from "You see a tall man standing in the street" to "You see Bob standing in the street".

One further point; note that with both the `mavis` object and the `bob` object we defined the initial location of the actor with the `@` symbol in the template. If we're defining an NPC of any complexity (for whom we're going to define quite a few associated objects) it's probably better to put all the code relating to that NPC in a separate source file, rather than nesting it all in the NPC's starting location with the + notation (which, with an NPC of any complexity, very quickly becomes ++, +++ and ++++).

We can give an NPC possessions and clothing just like the player character, by locating them just inside the relevant actor object. We can also give an NPC body parts (if we feel we need to) by making them components of the NPC. For example, immediately following the above definition of the bob object we might add:

```
+ Component 'thin moustache'
     name = (bob.theName) + '\'s moustache')
;

+ Wearable 'smart business suit*suits clothes'  'suit'
   wornBy = bob
;

+ stick: Thing 'walking stick*sticks'  'walking stick'
;
```

We'd no doubt also want to add descriptions for all three of these objects, and maybe some custom messages (e.g. responding to commands like **pull moustache**), but the minimalist code above suffices to demonstrate the principle.

14.3 Actor States

NPCs who show any interesting signs of life are likely to be doing different things at different times. Aunt Mavis may stand staring at herself in the mirror, or sit to read a book, or nod off to sleep, or engage in vigorous conversation with the player character. Bob won't stand around in the street forever, he may go into a restaurant to buy lunch, or in another scene he may be seated behind his desk or out playing golf. The way we want to describe an NPC, and the way we want NPCs to respond to what's going on around them, will vary according to what the NPC is up to at the time. To encapsulate this behaviour we use ActorState objects, which we locate in the actor object to which they refer. For example:

```
bob: Person 'tall man*men'  'tall man' @highStreet
   "He's a tall man, wearing a smart business suit and sporting a thin
     moustache. "
   isHim = true
   globalParamName = 'bob'
;

+ bobStanding: ActorState
   isInitState = true
   specialDesc = "{The bob/he} is standing in the street, looking in a shop
    window. "
   stateDesc = "He's looking in a shop window. "
;

+ bobWalking: ActorState
   specialDesc = "{The bob/he} is walking briskly down the street. "
   stateDesc = "He's walking briskly down the street. "
;
```

The most commonly used properties and methods defined on the ActorState class include:

- **isInitState** – set to true if this is the ActorState the associated actor starts out in.
- **specialDesc** – the description of the NPC as it appears in a room description when the NPC is in this ActorState.
- **stateDesc** – an additional description of the NPC appended to the desc property of the associated actor when the NPC is in this ActorState.
- **getActor()** - the actor with which this ActorState is associated (note, this should be treated as a read-only method; we don't use it to associate an Actor with an ActorState but only to find out which Actor is already associated with a particular ActorState.
- **activateState(actor, oldState)** - this method is executed just as this ActorState becomes active (i.e. becomes the current state for the associated Actor).
- **deactivateState(actor, newState)** – called just as the associated Actor is about to switch from this state to *newState*.

In addition ActorState defines the methods **beforeAction()**, **afterAction()**, **beforeTravel(traveler, connector)** and **afterTravel(traveler, connector)**, which have the same meaning as these methods do in sections 13.4 and 13.5 above, except that they are particular to the ActorState. This allows us to define a different reaction to actions and travel on each ActorState. If we want a common reaction (e.g. Aunt Mavis reacts the same way to the player character yelling no matter what ActorState she's in) we can define it on the actor object, but we must then call the inherited method, otherwise we'll break the mechanism that farms these responses out to ActorStates in other cases:

```
mavis: Person 'old frail aunt woman/mavis*women'  'Aunt Mavis' @lounge
   "Well past her prime, she is now looking distinctly frail. "
   afterAction()
   {
      if(gActionIs(Yell))
        "Aunt Mavis glowers at you with a look fit to freeze the sun. ";

      inherited; // DON'T FORGET THIS
   }
;
```

More usually, we'd define this kind of reaction on the ActorState, for example:

```
+ mavisReading: ActorState
    specialDesc = "Aunt Mavis is sitting in her favourite chair, engrossed
```

```
      in <i>The Last Chronicle of Barset</i>. "
    stateDesc = "She's sitting reading her favourite Trollope novel. "
    afterAction()
    {
       if(gActionIs(Yell))
         "Aunt Mavis peers over the top of her novel to give you a look
           that would have silenced even Mrs Proudie. ";
    }
;
```

Or

```
+ bobStanding: ActorState
   isInitState = true
   specialDesc = "{The bob/he} is standing in the street, looking in a shop
    window. "
   stateDesc = "He's looking in a shop window. "
   afterTravel(traveler, connector)
   {
      inherited(traveler, connector);

      if(traveler == me)
      {
         "{The bob/he} takes one look at you, and then turns away and
          starts walking briskly down the street. ";
         bob.setCurState(bobWalking);
      }
   }
;
```

Note the use of **setCurState(state)** to change an actor's ActorState to *state*. If we want to change an actor's ActorState in our code, we should always use this method, and never directly assign a value directly to the **curState** property. We can, however, of course test the **curState** property to find out what ActorState an actor is currently in.

ActorState defines a number of other methods, but one other of particular interest is **takeTurn()**. This is run each turn the associated actor is in this state, and does a number of things by default (so that if we override it we must be sure to call the inherited method unless we're really absolutely sure we don't want it). One of the things it does is call the **doScript()** method of the ActorState if the ActorState is also an EventList of some kind (provided the actor isn't already engaged in something with a higher priority like an AgendaItem or Conversation Node, on which see below) . This means we can can define an ActorState like this:

```
+ mavisReading: ActorState, ShuffledEventList
    specialDesc = "Aunt Mavis is sitting in her favourite chair, engrossed
      in <i>The Last Chronicle of Barset</i>. "
    stateDesc = "She's sitting reading her favourite Trollope novel. "
    eventList =
    [
       'Aunt Mavis turns over another page of her book. ',
       'Aunt Mavis chuckles to herself. ',
       'Aunt Mavis snorts with disapproval. ',
       'Aunt Mavis glances over the top of her book at you, as if to
```

```
        reassure herself that you\'re not disbehaving. '

    ]
;
```

And we'll see one of these 'Aunt Mavis' messages each turn, thus helping to bring Aunt Mavis a little more to life.

We can use the ActorState class (and it's sometimes useful to do so), but there are also a number of commonly used subclasses of ActorState with more specialized uses:

- **`HermitActorState`** – a state to use for an actor who is sleeping, unconscious, or so preoccupied with what he or she is doing that s/he won't respond to the player character when the player tries to address a conversational command to him or her. The **`noResponse`** property of a **`HermitActorState`** can be overridden (with a double-quoted string) to display a message explaining why the actor won't respond.
- **`AccompanyingState`** – a state to use for an actor who will follow the player character around for as long as s/he's in this state. If you override the **`beforeTravel()`** method on an **`AccompanyingState`** be absolutely sure to call the inherited method, or you'll disable the following mechanism! The **`accompanyTravel(traveler, conn)`** can be used to check whether the actor is willing to follow *traveler* (normally the player character) through the connector *conn;* return nil to stop the actor following the *traveler* that way, or use **`exit`** to prevent the *traveler* going that way too. Override **`arrivingWithDesc`** to display a message describing the arrival of the following actor in a new location.
- **`GuidedTourState`** – a subclass of **`AccompanyingState`** to use when you want the actor to lead the player character. The actor will wait for the player character to follow him or her, and the player can choose not to follow. Set the **`escortDest`** property to the TravelConnector the actor wants to lead the player character through. Set the **`stateAfterEscort`** property to the ActorState object to change to after the player character has followed the actor through this connector (this might be another **`GuidedTourState`** representing the next stage of the journey on which the actor wants to lead the player character). If you use this class you might want to add **`TourGuide`** to the class list of the associated actor; this will allow the player to use the command **follow x** to follow the actor x when he or she is in a **`GuidedTourState`**.
- **`ConversationReadyState`** – a state to use for an NPC who is ready to enter into a conversation when greeted. We'll explain this further below in section 14.6.
- **`InConversationState`** – a state to use for an NPC who was in a **`ConversationReadyState`** but is now conversing with the player character. We'll explain this further below.

14.4 Conversing with NPCs – Topic Entries

The standard form of conversation implemented in the TADS 3 library is the ask/tell model. This handles commands like **ask bob about shopping** or **tell mavis about barchester**. It also handles commands of the form **show gold ring to bob**, **give coin to shopkeeper** and **ask king henry for his crown**.

In TADS 3, the responses to such conversational commands are defined in objects of the `TopicEntry` class, or rather, of one of the various subclasses of `TopicEntry`. We have already met one such subclass, namely `ConsultTopic`, used to look up entries in a `Consultable`. The other `TopicEntry` subclasses work in a similar way, except that they handle conversational commands (of the types we have just seen) rather than attempts to look things up.

The various subclasses of TopicEntry we are immediately concerned with here are:

- `AskTopic` – the response to a command of the form **ask someone about x**.
- `TellTopic` – the response to a command of the form **tell someone about x**.
- `AskTellTopic` – responds to **ask someone about x** or **tell someone about x**.
- `GiveTopic` – responds to **give something to someone**
- `ShowTopic` – responds to **show something to someone**
- `GiveShowTopic` – responds to **give something to someone** or **show something to someone**
- `AskForTopic` – responds to **ask someone for x**
- `AskAboutForTopic` – responds to **ask someone about x** or **ask someone for x**
- `AskTellAboutForTopic` – responds to **ask someone about x** or **tell someone about x** or **ask someone for x**
- `AskTellGiveShowTopic` – responds to **ask someone about something** or **tell someone about something** or **give something to someone** or **show something to someone**.
- `AskTellShowTopic –` responds to **ask someone about something** or **tell someone about something** or **show something to someone**

In the above list, **someone** is the NPC we're talking with, **something** is generally a Thing (i.e. an object of class Thing implemented somewhere in the game) and **x** can be either a Thing or a Topic.

To define *which* Thing or Topic a TopicEntry relates to we define its `matchObj` property. This can either be a single object (a Thing, or where appropriate, a Topic) or it can be a list of Things (or, where appropriate, Topics, or a mixture of Things and Topics). If it's a list then the TopicEntry will be matched provided any one of the objects in the list match what the player wants to talk about, and provided the player character knows about the object in question (i.e. `gPlayerChar.knowsAbout(x)` is true, where x is the relevant object).

How the NPC responds to the conversational command (or better, the complete conversational exchange) is defined in the `topicResponse` property. This may be defined simply as a double-quoted string to display what is said, or it can be defined as a method to display the conversational command and carry out some related side-effects, e.g.:

```
+ GiveTopic
     matchObj = coin
     topicResponse()
     {
         "You give {the bob/him} the coin and he accepts it with a curt nod. ";
          coin.moveInto(bob);
     }
;
```

Or in a simpler case:

```
+ AskTopic
     matchObject = tTrollope
     topicResponse = "<q>Tell me, aunt, do you really think Anthony Trollope is
       such a great novelist?</q> you ask.\b
     <q>I find his writing infinitely preferable to your idle chatter,</q> she
     replies frostily. "
;
```

Since the `matchObj` and `topicResponse` properties of Topic Entries are defined so frequently, the above examples can be written more succinctly using a template:

```
+ GiveTopic @coin
     topicResponse()
     {
         "You give {the bob/him} the coin and he accepts it with a curt nod. ";
          coin.moveInto(bob);
     }
;

+ AskTopic @tTrollope
     "<q>Tell me, aunt, do you really think Anthony Trollope is
       such a great novelist?</q> you ask.\b
     <q>I find his writing infinitely preferable to your idle chatter,</q> she
     replies frostily. "
;
```

We can also use a template when the matchObj contains a list of objects, for example if the matchObj for the AskTopic had been:

```
matchObj = [tTrollope, novel]
```

We could define the AskTopic as:

```
+ AskTopic [tTrollope, novel]
     "<q>Tell me, aunt, do you really think Anthony Trollope is
       such a great novelist?</q> you ask.\b
      <q>I find his writing infinitely preferable to your idle chatter,</q> she
      replies frostily. "
;
```

One slight problem with this AskTopic is that if the player keeps issuing the command **ask mavis about trollope** or **ask mavis about novel** she'll keep giving the same response, which will pretty quickly make her seem either surreal or robotic. If we want to vary her response, we can include an EventList class in the class list of the TopicEntry and define the eventList property instead of the topicResponse property, for example:

```
+ AskTopic, StopEventList
    matchObj = [tTrollope, novel]
    eventList =
    [
      '<q>Tell me, aunt, do you really think Anthony Trollope is
       such a great novelist?</q> you ask.\b
       <q>I find his writing infinitely preferable to your idle chatter,</q> she
       replies frostily.',

      '<q>Seriously, aunt, is Trollope that great a novelist?</q> you ask,
       <q> My English master at school always thought him a bit of a
       second-rate Victorian hack.</q>\b
       <q>Then you obviously went to the wrong school!</q> your aunt
       replies, visibly bridling. ',

      'It might be wise to leave that topic alone, before you cause any
       further offence. '
    ]
;
```

Once again, this is sufficiently common that we can write it using a template:

```
+ AskTopic, StopEventList [tTrollope, novel]
    [
      '<q>Tell me, aunt, do you really think Anthony Trollope is
       such a great novelist?</q> you ask.\b
       <q>I find his writing infinitely preferable to your idle chatter,</q> she
       replies frostily.',

      '<q>Seriously, aunt, is Trollope that great a novelist?</q> you ask,
       <q> My English master at school always thought him a bit of a
       second-rate Victorian hack.</q>\b
       <q>Then you evidently went to the wrong school!</q> your aunt
       declares, visibly bridling. ',

      'It might be wise to leave that topic alone, before you cause any
       further offence. '
    ]
;
```

We could use a similar template form with `@tTrollope` (a single `matchObj`) in place of `[tTrollope, novel]` (the `matchObj` list).

Another way to vary the response is to make the availability of certain Topic Entries dependent upon some condition, such as what has been said previously, or some aspect of the game state. We can do that by attaching a condition (or rather an expression that may be either true or false) to the `isActive` property of a Topic Entry. For example, we may want to ask Bob a certain question about the lighthouse only if the player character has actually seen the lighthouse, so we might write:

```
+ AskTopic @lighthouse
  "<q>What exactly happened at the lighthouse?</q> you ask, <q>When I saw
   it, it looked as if someone had tried to set it on fire!</q>\b
   <q>It was the troubles,</q> he replies grimly, <q>but you don't want to
   know about them, you really don't!</q> "

   isActive = me.hasSeen(lighthouse)
;
```

It may that we'd want to ask a different question about the lighthouse if the player character hadn't seen it, so we *could* write:

```
+ AskTopic @lighthouse
  "<q>What's this I hear about a lighthouse?</q> you ask.\b
   <q>Oh, you don't want to go there,</q> he tells you, <q>there's nothing
   of interest at all, besides... well, just don't bother with it, that's
   all.</q> "

   isActive = !me.hasSeen(lighthouse)
;
```

Then we'd get one response when the player character had seen the lighthouse and another when s/he hadn't.

An alternative would be to make use of a property we haven't mentioned yet, namely `matchScore`. When TADS 3 finds more than one TopicEntry that could match the conversational command the player typed, it chooses the one with the highest matchScore. The default matchScore of a TopicEntry is 100, so we could write:

```
+ AskTopic @lighthouse
  "<q>What's this I hear about a lighthouse?</q> you ask.\b
   <q>Oh, you don't want to go there,</q> he tells you, <q>there's nothing
   of interest at all, besides... well, just don't bother with it, that's
   all.</q> "
   matchScore = 90
;
```

The `matchScore` property can also be assigned via the template, using the + symbol and making it the first item, so this could be written:

```
+ AskTopic +90 @lighthouse
  "<q>What's this I hear about a lighthouse?</q> you ask.\b
   <q>Oh, you don't want to go there,</q> he tells you, <q>there's nothing
   of interest at all, besides... well, just don't bother with it, that's
   all.</q> "
;
```

When the player character hasn't seen the lighthouse the first AskTopic can't match (since its **isActive** property evaluates to nil), so we get the second AskTopic response. When the player character has seen the lighthouse both AskTopics match, so the one with the higher **matchScore** wins, and we get the "What exactly happened at the lighthouse?" exchange.

Although **matchScore** can be useful in other circumstances, and this would work here, in practice we'd probably use a different technique for this particular kind of situation, using an **AltTopic**. An AltTopic is an Alternative Topic Entry for use when we want to match the same object, or set of objects, but want a different response when some particular condition is true. In this instance we'd probably set it up like this:

```
+ AskTopic @lighthouse
  "<q>What's this I hear about a lighthouse?</q> you ask.\b
   <q>Oh, you don't want to go there,</q> he tells you, <q>there's nothing
   of interest at all, besides... well, just don't bother with it, that's
   all.</q> "
;

++ AltTopic
   "<q>What exactly happened at the lighthouse?</q> you ask, <q>When I saw
   it, it looked as if someone had tried to set it on fire!</q>\b
   <q>It was the troubles,</q> he replies grimly, <q>but you don't want to
   know about them, you really don't!</q><.reveal bob-troubles> "

   isActive = me.hasSeen(lighthouse)
;
```

Note how this works. We locate the **AltTopic** in the Topic Entry with which it's associated (by giving it one more **+** than its associated Topic Entry has in the containment hierarchy). We don't define the objects it's meant to match, or whether it's meant to match ask, tell, show, or give, since these will always be the same as for the Topic Entry we've located it in. Instead we define its **isActive** property to give the condition that must be true for this **AltTopic** to be used in preference to its parent Topic Entry. We can follow a Topic Entry with as many AltTopics (located in it) as we like; the one that will be used is always the last one in the list whose **isActive** property is true.

Note also that we've added **<.reveal bob-troubles>** to the topicResponse of the AltTopic. We encountered the reveal mechanism in the chapter on Knowledge, but this is the first time we've seen it used in the situation for which it was principally designed, namely to keep track of what has already been said in a conversation. Just to recap, including **<.reveal *tag*>** in a string cause the string *tag* to be added to the table of things that have been revealed, which we can then test for with **gRevealed('tag')**. We've used the tag 'bob-troubles' here to note that Bob has now mentioned the troubles. This would allows us to write a subsequent AskTopic that should only come into effect once Bob has referred to the troubles in something he's said:

```
+ AskTopic @tTroubles
  "<q>What are these troubles you mentioned?</q> you want to know.\b
   <q>Never you mind, they're best forgotten,</q> he mutters. "
   isActive = gRevealed('bob-troubles')
;
```

This ensures that the question about the troubles can't be asked until Bob has mentioned the troubles (since the exchange clearly presupposes that Bob *has* previously mentioned the troubles). Of course it doesn't stop the player from typing the command **ask bob about troubles**, but it may be that until Bob mentions the troubles there won't be any matching Topic Entry for this question. Indeed players are likely to try asking and telling our NPCs about all sorts of things for which we haven't provided a specific response. In such a case we ideally need our NPCs to make some vague non-committal response that shows that they're still in the conversation without saying anything positively incongruous. For this purpose TADS defines a special kind of Topic Entry called a `DefaultTopic`. Or rather, TADS defines a range of DefaultTopic classes to handle default responses for a variety of conversational commands:

- `DefaultAskTopic` – responds to **ask about**
- `DefaultTellTopic` – responds to **tell about**
- `DefaultAskTellTopic` – responds to **ask about** or **tell about**
- `DefaultGiveTopic` – responds to **give**
- `DefaultShowTopic` – responds to **show**
- `DefaultGiveShowTopic` – responds to **give** or **show**
- `DefaultAskForTopic` – responds to **ask for**
- `DefaultAnyTopic` – responds to any conversational command

These various kinds of `DefaultTopic` match any topic or object, but they have low matchScores, so that where a more specific response exists (and is active), it will always be used in preference to the DefaultTopic. There's also a hierarchy among the `DefaultTopic` classes: a `DefaultAnyTopic` has a `matchScore` of 1; `DefaultAskTellTopic` and `DefaultGiveShowTopic` have a `matchScore` of 2; and the other four have a `matchScore` of 3. This means, for example, that if we have defined a `DefaultAnyTopic`, a `DefaultAskTellTopic`, and a `DefaultAskTopic`, the `DefaultAskTellTopic` will be used in preference to the `DefaultAnyTopic`, and the `DefaultAskTopic` in preference to the `DefaultAskTellTopic`.

Normally the only property we need to define on a DefaultTopic is its `topicResponse`. So, for example, for Aunt Mavis we might define:

```
+ DefaultGiveShowTopic
     topicResponse = "Aunt Mavis waves {the dobj/him} away with an impatient
      gesture. "
;

+ DefaultAnyTopic
     topicResponse = "Aunt Mavis peers over the top of her book and replies,
       <q>Why people feel the need to fill the air with such pointless noise
       is quite beyond me. Really, if you don't have anything more important
       to talk about than that, you should not disturb me with it!</q> "
;
```

Once again, we can make these definitions a bit more concise using a template:

```
+ DefaultGiveShowTopic
     "Aunt Mavis waves {the dobj/him} away with an impatient gesture. "
;

+ DefaultAnyTopic
     "Aunt Mavis peers over the top of her book and replies,
     <q>Why people feel the need to fill the air with such pointless noise
     is quite beyond me. Really, if you don't have anything more important
     to talk about than that, you should not disturb me with it!</q> "
;
```

Since players are likely to encounter our DefaultTopics fairly frequently, it’s a good idea to vary the response they’ll see; once again, this can help to make our NPCs seem a little less robotic. The way to do this is to add an EventList class (usually ShuffledEventList, in this context) to the DefaultTopic class and define a varied list of default responses in the `eventList` property. This property can again be implicitly defined using the DefaultTopic template, e.g.

```
+ DefaultAnyTopic, ShuffledEventList
  [
     'Aunt Mavis peers over the top of her book and replies,
     <q>Why people feel the need to fill the air with such pointless noise
     is quite beyond me. Really, if you don't have anything more important
     to talk about than that, you should not disturb me with it!</q> ',

     '<q>Can't you see I'm trying to read?</q> she complains irritably,
      <q>Really, the manners of people these days!</q> ',

     '<q>We can discuss that when I'm not trying to read,</q> she suggests. '
  ]
;
```

We’ve now covered the basics of using Topic Entries to define conversational responses apart from one rather major point: we’ve shown how to define Topic Entry objects, but we haven’t yet discussed where to put them. Clearly Topic Entries need to be associated with the actor whose conversation they’re implementing, and this can be done in one of three ways:

1. Topic Entries can be located directly in their associated actor, in which case (with certain qualifications) they'll be available whenever the player character addresses that actor.
2. Topic Entries can be located in one of their associated actor's ActorStates (typically, but not exclusively, an **InConversationState**), in which case they will be available only when the actor is in that ActorState.
3. TopicEntries can be located in a **TopicGroup.** They are then available when the **TopicGroup** is active (and its location is available).

A **TopicGroup** is basically a way to apply a common **isActive** condition to a group of Topic Entries. Topic Entries within a **TopicGroup** may also define their own individual **isActive** conditions, in which case both the **isActive** condition on the **TopicGroup** and the **isActive** condition on the individual Topic Entry must be true for that individual Topic Entry to be reachable. A TopicGroup can go anywhere a Topic Entry can go, located either in an Actor, or in an ActorState, or in another TopicGroup. In addition to the **isActive** property, TopicGroup defines a **matchScoreAdjustment** property which can be used to boost the matchScores of all the Topic Entries in the TopicGroup. For example, if we give a TopicGroup a **matchScoreAdjustment** of 10, then any Topic Entry within it which has a default **matchScore** of 100 will have an effective **matchScore** of 110. Apart from the effects of the TopicGroup's **isActive** and **matchScoreAdjustment** properties, Topic Entries in a TopicGroup behave just as if they were in that TopicGroup's container.

This may all become a little clearer with a skeletal example:

```
mary: Person 'mary/woman*women' 'Mary'
   isProperName = true
   isHer = true
;

+ TopicGroup
    isActive = (mary.curState is in (maryWalking, maryTalking))
;

++ AskTopic @robert
   "blah blah"
;

++ TellTopic @tWedding
   "blah blah"
;

+ AskTopic @mary
  "blah blah"
;

+ maryWalking: AccompanyingState
  specialDesc = "Mary is walking along beside you. "
;
```

```
++ AskTopic @tShopping
   "blah blah"
;

++ TellTopic @robert
  "blah blah"
;

+++ AltTopic
  "blah blah"
  isActive = gRevealed('robert-affair')
;

++ DefaultAnyTopic
  "blah blah"
;

+ maryTalking: ActorState
  specialDesc = "Mary is looking at you. "
;

++ GiveTopic @ring
   "blah blah"
;

++ AskTopic @robert
  "blah blah"
;

+ marySinging: HermitActorState
  specialDesc = "Mary is busily rehearsing an aria from <i>The Marriage of
   Figaro<i>. "
  noResponse = "You don't like to interrupt her singing. "
;
```

The TopicGroup directly under Mary is active when Mary is either in the **maryTalking** state or in the **maryWalking** state; this is a convenient way to make a group of Topic Entries common to two or more Actor States (but not all of them). Note that there's an AskTopic for Robert defined under both the TopicGroup and the **maryTalking** ActorState; when Mary is in the **maryTalking** state it’s the latter that will be used, since an ActorState’s Topic Entries are always used in preference to that of an Actor’s. A subtler effect of this is that the DefaultAnyTopic in the **maryWalking** state will render all the Topic Entries in our TopicGroup unreachable when Mary is in the **maryWalking** state. This probably isn’t what we want; the solution (if we see this is a problem) is to override either the **excludeMatch** property or the **deferToEntry(other)** method of the DefaultAnyTopic.

The **excludeMatch** property can contain a list of topics we don’t want a DefaultTcpic to match, so if want the two Topic Entries we’ve put in the TopicGroup and the one we’ve put directly in the **mary** object to be reachable even when when Mary is in the **maryWalkingState** we could define the **DefaultAnyTopic** as:

```
++ DefaultAnyTopic
  "blah blah"
  excludeMatch = [mary, robert, tWedding]
;
```

A more general solution, which doesn't rely on our having to list specific topics to exclude from what a DefaultTopic matches is to override the `deferToEntry(other)` method to have the DefaultTopic defer to any Topic Entry outside the current ActorState (or ConvNode). This will occur for any Topic Entry *other* for which `deferToEntry(other)` returns true. So, to make all the non-default Topic Entries defined on the actor object (or any TopicGroups located in the actor object) available even when Mary is in `maryWalkingState` we should define the `DefaultAnyTopic` as:

```
++ DefaultAnyTopic
  "blah blah"
  deferToEntry(other)  { return !other.ofKind(DefaultTopic); }
;
```

14.5 Suggesting Topics of Conversation

One thing we don't want is for people playing our game to feel that they're having to play "guess the topic" when they're conversing with our NPCs. We don't want them to become frustrated by reading our default responses dozens of times over while hunting for the few topics we've actually implemented, and we don't want them to miss the topics that are vital to their understanding of the game or the advancement of the plot. It may be that we can avoid all these problems by making it obvious from the context and from our NPCs' previous replies which topics are worth talking about, but it may be we want to give our players a helping hand by suggesting which topics are particularly worth asking or telling about.

We can do this using *SuggestedTopics*. Those that correspond to the Topic Entry types we have met so far come in the following flavours:

- `SuggestedAskTopic` – suggest something to ask about
- `SuggestedTellTopic` – suggest something to tell about
- `SuggestedGiveTopic` – suggest something to give
- `SuggestedShowTopic` – suggest something to show
- `SuggestedAskForTopic` – suggest something to ask for

The way we use these is to add them to the class list of the Topic Entries we want to suggest, and then to define a `name` property defining how we want the suggestion described. This should be a (single-quoted) string that could meaningfully complete a sentence like "You could ask Anne about..." or "You could tell Bob about..." or "You could show him..." For example:

```
+ AskTopic, SuggestedAskTopic @mavis
  "<q>How are you today, Aunt Mavis?</q> you enquire.\b
   <q>Well enough,</q> she replies. "
  name = 'herself'
;

+ TellTopic, SuggestedTellTopic @me
  "<q>You know aunt, I've been meaning to tell you about..."
  name = 'yourself'
;

+ GiveShowTopic, SuggestedShowTopic @ring
  "<q>That's a nice ring!</q> she declares. "
  name = (ring.theName)
;

+ AskForTopic, SuggestedAskForTopic @tMoney
  "<q>Could you lend me..."
  name = 'money'
;
```

Suggestions are displayed either when the player character explicitly greets the actor (with a command like **talk to aunt** or **say hello to mavis**) or in response to an explicit **topics** command, or when the game author schedules a display of suggested topics using the `<.topics>` tag in conversational output. Suggestions are generally only displayed when they're reachable (although occasionally there can be arrangements of Topic Entries that are too complicated for the library to work this out). Normally each SuggestedTopic will only be suggested once, that is only suggested until the player tries conversing about the topic in question for the first time. Strictly speaking, a SuggestedTopic continues to be suggested until its `curiositySatisfied` property becomes true. By default this is when it has been accessed the number of times defined in its `timesToSuggest` property, and in turn the default value of `timesToSuggest` is 1. But when a SuggestedTopic is combined with an EventList, we may want to suggest the topic more than once. Consider the following example:

```
+ AskTopic, StopEventList, SuggestedAskTopic [tTrollope, novel]
    [
      '<q>Tell me, aunt, do you really think Anthony Trollope is
       such a great novelist?</q> you ask.\b
       <q>I find his writing infinitely preferable to your idle chatter,</q> she
       replies frostily. ',

      '<q>Seriously, aunt, is Trollope that great a novelist?</q> you ask,
       <q> My English master at school always thought him a bit of a
       second-rate Victorian hack.</q>\b
       <q>Then you evidently went to the wrong school!</q> your aunt
       declares, bridling visibly. ',

      'It might be wise to leave that topic alone, before you cause any
       further offence. '
    ]
    name = 'Anthony Trollope'
    timesToSuggest = 2
    isConversational = (!curiositySatisfied)
;
```

Here it seems sensible to change timesToSuggest to 2, since there are two potentially interesting responses. There's no point in suggesting this topic for the third time, however, since the third response is simply a way of saying "this topic is exhausted". At the same time it seems a good idea to define isConversational to return nil once the final response is reached (which is also when curiosity is satisfied), since this third response is indeed not conversational: it doesn't represent a conversational exchange, it simply tells the player why no further conversational exchange on that topic should take place. The practical effect of this is that asking Aunt Mavis about Trollope for the third time won't trigger greeting protocols (which we'll explain just below), which aren't appropriate when no conversation takes place; in essence, we want to avoid this kind of thing:

```
>ask aunt about trollope
"Hello there, Aunt!" you declare enthusiastically.

"Oh, it's you again," she replies without enthusiasm.

It might be wise to leave that topic alone, before you cause any further offence.
```

One further refinement is the use of SuggestedTopics with AltTopics. Normally, an AltTopic would be regarded as distinct from its parent topic from the point of view of topic suggestions. Consider the following example:

```
+ AskTopic, SuggestedAskTopic @ring
  "<q>Have you heard anything about a ring - a special ring?</q> you ask.\b
   <q>I've heard talk of a magic gold ring - but it's only talk,</q> he
   replies.
   name = (ring.theName)
;

++ AltTopic, SuggestedAskTopic
  "<q>I've seen a gold ring, and there seemed something quite strange about
   it. Could that be the magic gold ring everyone's talking about?</q>\b
  <q>It's probably just an ordinary gold ring,</q> he tells you.
   name = (ring.theName)
   isActive = me.hasSeen(ring)
;
```

In this case the ring will be suggested as something to ask about until the player asks about it once. After the player character has seen the ring, the ring topic suggestion will once again be displayed until it has been asked about again. If we only wanted the ring to be suggested once, regardless of whether the player character asks about it before or after seeing it, we can use the SuggestedTopicTree class. This effectively makes the entire group of AltTopics, together with its parent Topic Entry, into a single suggestion. We use SuggestedTopicTree by mixing it in with the class list of the parent Topic Entry, like this:

```
+ AskTopic, SuggestedTopicTree, SuggestedAskTopic @ring
  "<q>Have you heard anything about a ring - a special ring?</q> you ask.\b
```

```
   <q>I've heard talk of a magic gold ring - but it's only talk,</q> he
   replies.
   name = (ring.theName)
;

++ AltTopic
  "<q>I've seen a gold ring, and there seemed something quite strange about
   it. Could that be the magic gold ring everyone's talking about?</q>\b
  <q>It's probably just an ordinary gold ring,</q> he tells you.
   isActive = me.hasSeen(ring)
;
```

SuggestedTopics are easy enough to use (provide we remember to define their `name` property, which it's very easy to forget to do); what's more difficult is to decide which topics to suggest. If we've defined a large number of Topic Entries, perhaps to cover all the things our beta-testers tried to converse with our NPCs about, making all of them SuggestedTopics would probably be overwhelming for our players. SuggestedTopics can be an aid to players, but they can also become a hindrance if they suggest a huge list of topics players think they have to work through. It's often best, then, to focus on suggesting only those topics that are most essential to the plot, whether through providing information for solving puzzles or in terms of having particularly interesting responses.

14.6 Hello and Goodbye – Greeting Protocols

In real life, people don't generally leap straight into the middle of a conversation and then break it off arbitrarily; but in Interactive Fiction conversations can all too easily be like that. TADS 3 tries to avoid this by implementing a scheme of greeting protocols. Not only does this make it possible to begin and end a conversation by saying hello and goodbye in response to explicit player commands, it allows these greeting (and farewell) protocols to be triggered implicitly whenever the player character starts and ends a conversation with an NPC.

For this to work, we need to set up a pair of ActorStates for the NPC to switch between as the conversation begins and ends. One of these must be a `ConversationReadyState`, and the other an `InConversationState`. The NPC starts off in the `ConversationReadyState`. When the player character addresses a greeting or any other conversation to the NPC, the NPC switches to the `InConversationState`, and any greeting message we have defined is displayed. When the conversation comes to an end for whatever reason (and there are several possible reasons) the NPC reverts back to the `ConversationReadyState` (unless we define some other state for the NPC to change to).

The `InConversationState` that a `ConversationReadyState` switches to is defined in its `inConvState` property; otherwise a `ConversationReadyState` is defined using more or less the same properties as we'd use for an ordinary `ActorState`.

`InConversationState` also inherits the standard methods and properties of

ActorState, but also defines a few new ones and overrides some of those that it inherits:

- **attentionSpan –** this is the number of turns the NPC will remain in this InConversationState before becoming bored waiting for the player character to speak (if the player fails to enter any conversational commands for that number of turns). The default value is 4, but we can easily override that if we want our NPC to have a shorter or longer attentionSpan. If we want out NPC to have an infinite attentionSpan when in this state (i.e., the conversation will never be ended on account of the NPC's boredom), we should set this property to nil.
- **nextState** – the ActorState to switch to when the conversation is concluded. By default this is the value of the **lastState** property, which is automatically set to the value of the ActorState the NPC was previously in (provided it was a ConversationReadyState) before switching to this state. By default, then, at the end of the conversation the NPC will switch back to the ConversationReadyState it was in before.

Rather than having to set the **inConvState** property on a ConversationReadyState explicitly, we can locate the ConversationReadyState within the InConversationState with which we wish to associate it in order to have the library make the association for us; this also makes it a bit easier to see the pairing of the states in the code; for example:

```
bob: Person 'tall thin man/bob*men' 'Bob'
   "He's a tall, thin man. "
   isHim = true
   isProperName = true
;


+ bobTalking: InConversationState
   attentionSpan = 5
   specialDesc = "Bob is standing by the shop window, waiting for you to
    speak. "
   stateDesc = "He's waiting for you to speak. "
;

++ bobLooking: ConversationReadyState
   isInitState = true
   commonDesc = " standing in the street, peering into a shop window. "
   specialDesc = "Bob is <<commonDesc>>"
   stateDesc = "He's <<commonDesc>>"
;
```

Note that in this example, **commonDesc** isn't a standard library property, it's simply a custom property we've defined here to avoid having to type the same text for the **specialDesc** and **stateDesc** properties.

The next stage is to define some more Topic Entries to make Bob say Hello and Goodbye at the appropriate moments. To do this we can use one or more of the following Topic Entry classes:

- `HelloTopic` – A response to an explicit greeting; also used for an implicit greeting response if no `ImpHelloTopic` has been defined.
- `ImpHelloTopic` – response to an implicit greeting
- `ByeTopic` – A response to an explicit farewell; also used for an implicit farewell response if no implicit response has been defined.
- `ImpByeTopic` – response to an implicit farewell if the relevant more specialized implicit farewell responses (one of the next three classes) hasn't been defined.
- `BoredByeTopic` – an implicit farewell response used when the NPC becomes bored waiting for the player character to speak (i.e. the NPC's attentionSpan has been exceeded)
- `LeaveByeTopic` – an implicit farewell response used when the player character terminates the conversation by leaving the vicinity
- `ActorByeTopic` – a farewell response for the case in which the NPC decides to terminate the conversation.
- `HelloGoodbyeTopic` – response to either HELLO or GOODBYE

An *explicit* greeting or farewell is one in which the player explicitly types a command such as **talk to bob** or **bob, hello** or **bye**. An *implicit* greeting is one triggered by the player issuing a conversational command (such as **ask bob about shop**) without first issuing an explicit greeting. An *implicit* farewell is one triggered by ending the conversation other than by an explicit **bye** or **say goodbye** command (or the like).

To make the actor terminate the conversation we can simply call `endConversation()` on the actor object, e.g.:

```
bob.endConversation();
```

All these various HelloTopics and ByeTopics should be located in the ConversationReadyState. The Topic Entries relating to the ongoing conversation should go in the InConversationState. So, to expand our previous example, we might have:

```
bob: Person 'tall thin man/bob*men' 'Bob'
   "He's a tall, thin man. "
   isHim = true
   isProperName = true
;

+ bobTalking: InConversationState
   attentionSpan = 5
   specialDesc = "Bob is standing by the shop window, waiting for you to
    speak. "
   stateDesc = "He's waiting for you to speak. "
;

++ bobLooking: ConversationReadyState
   isInitState = true
   commonDesc = " standing in the street, peering into a shop window. "
   specialDesc = "Bob is <<commonDesc>>"
   stateDesc = "He's <<commonDesc>>"
;

+++ HelloTopic, StopEventList
  [
    '<q>Hello, there!</q> you say.\b
     <q>Hi!</q> Bob replies, turning to you with a smile. ',

    '<q>Hello, again,</q> you greet him.\b
     <q>Yes?</q> he replies, turning back to you. '
  ]
;

+++ ByeTopic
  "<q>Well, cheerio then!</q> you say.\b
   <q>'Bye for now,</q> Bob replies, turning back to the shop window.
;

+++ BoredByeTopic
   "Bob gives up waiting for you to speak and turns back to the shop window. "
;

+++ LeaveByeTopic
   "Bob watches you walk away, then turns back to the shop window. ";
;

+++ ActorByeTopic
   "<q>Goodness! Is that the time?</q> Bob declares, glancing at his watch,
    <q>I'd best be going! Goodbye!</q>\b
    So saying, he turns away and hurries off down the street. "
;

++ AskTopic @bob
   "<q>How are you today?</q> you ask.\b
    <q>Fine, just fine,</q> he assures you. "
;
```

Note that this final TopicEntry is back inside the InConversationState, where all our regular Topic Entries specific to this InConversationState should go.

14.7 Conversation Nodes

There come points in a conversation when a particular set of responses become appropriate that weren't appropriate before, and soon won't be appropriate again. Generally this happens when the other party to the conversation asks a question, or else makes a statement that demands (or invites) a particular type of response. If Bob asks "Would you like me to show you to the lighthouse?" it becomes relevant to reply **yes** or **no**, although neither of those responses would be appropriate if interjected into some random point in the conversation, and their significance would be somewhat altered if offered in response to a different question such as "Are you sleeping with my wife?".

To model such points in a conversation TADS 3 uses Conversation Nodes. These are objects of the **ConvNode** class. A ConvNode is a little like an ActorState or TopicGroup in that we can put a number of Topic Entries in it, but it is also a little different in function. We can also use one type of Topic Entry in a ConvNode that we can't use elsewhere: a **SpecialTopic**. There are also a couple of TopicEntry classes we can use elsewhere, but that are most likely to be useful in ConvNodes: **YesTopic** and **NoTopic**, which respond to **yes** and **no** respectively. If you want "yes" or "no" to be suggested as possible responses you can add **SuggestedYesTopic** and **SuggestedNoTopic** to the class list, as with other SuggestedTopic types.

At its simplest, the definition of a ConvNode object can be very simple indeed:

```
+ ConvNode
    name = 'node-name'
;
```

This can be made even more compact using the ConvNode template:

```
+ ConvNode 'node-name';
```

Here 'node-name' can be any string we like (so long as it doesn't contain the character '>'), but it must be unique among the names we give the ConvNodes for any particular actor. Following the ConvNode, and located within it, we put the Topic Entries that are relevant when the ConvNode is active:

```
+ ConvNode 'lighthouse';

++ YesTopic, SuggestedYesTopic
   "<q>Yes, I would like you to show me the lighthouse,</q> you say.\b
    <q>Right; I can't take you there now. Come back and meet me here at six,</q>
    he tells you. "
;

++ NoTopic, SuggestedNoTopic
  "<q>No, I've been warned that the lighthouse is not a good place to visit,</q>
   you reply.\b
   <q>Very well,</q> he shrugs. "
;
```

ConvNodes can be more complicated that this, but we'll look at the complications shortly. The next question to address is how we get the conversation into a particular ConvNode. There are basically three ways:

- We can call `setConvNode(node)` on the Actor object.
- We can use a `<.convnode node-name>` tag in the response of a Topic Entry.
- We can call `initiateConversation(state, node)` on the Actor object.

In both `setConvNode(node)` and `initiateConversation(state, node)`, the *node* parameter can be either the node object's identifier (if it's not an anonymous object) or the string assigned to its name property. For example, if we were wanting to switch to a ConvNode defined like this:

```
myNode: ConvNode 'node-name';
```

The *node* parameter could be specified either as `myNode` or as `'node-name'`. Since ConvNodes tend to be anonymous objects the second is likely to be more common.

The second way of switching to a ConvNode, using the `<.convnode >` tag, is typically used like this:

```
++ AskTopic @lighthouse
   "<q>What's this I hear about the lighthouse?</q> you ask.\b
    <q>It's easier to explain if you see it for yourself,</q> he replies,
    <q>would you like me to show you the lighthouse?</q><.convnode lighthouse> "
;
```

In this case, when we enter the ConvNode, the game has just displayed the question to which yes or no (the responses defined in the ConvNode) are the obvious possible answers.

The third way of entering a ConvNode, by calling `initiateConversation(state, node)` on the actor; *state* is either nil or the ActorState we want the actor to switch to. If it's nil, the actor either remains in its current ActorState or, if that ActorState is a ConversationReadyState, the actor will switch to the related InConversationState. We'd normally use this to get an NPC to start a conversation (although we can also use it to make an NPC steer an existing conversation in a new direction). So, for example, we could write:

```
bobStanding: ConversationReadyState
   specalDesc = "Bob is standing in the street, looking in a shop window. "
   afterTravel(traveler, connector)
   {
      inherited(traveler, connector);
      if(traveler == me)
         bob.initiateConversation(nil, 'lighthouse');
   }
;
```

If we enter the ConvNode in this fashion, there has been no previous conversation to which the player character might respond. In this case we can supply it in the `npcGreetingMsg` property of the ConvNode:

```
+ ConvNode 'lighthouse'
   npcGreetingMsg = "Seeing you approach, Bob turns to you and asks, <q>Hello,
     there! I hear you've been asking about the lighthouse. Would you like
     me to take you there?</q>"
;
```

If we thought we might want to enter this ConvNode more than once, we might want to vary the message that's displayed. We can do that by defining the `npcGreetingList` property instead, and attaching an EventList object to it:

```
+ ConvNode 'lighthouse'
   npcGreetingList: StopEventList
```

```
    {
     [
       'Seeing you approach, Bob turns to you and asks, <q>Hello,
        there! I hear you\'ve been asking about the lighthouse. Would you like
        me to take you there?</q> ',

       'Bob looks up at your approach and says, <q>Ah, you\'re back. Shall we
        go to the lighthouse now? </q>'
    ]
   }
;
```

All the ConvNodes we have seen so far would last only until the player makes some conversational response. As these examples have been defined, that response could be any conversational command, not just the yes or no Bob's question expects. If we want an NPC to insist on receiving a reply to his question, we have to do quite a bit more work. First, we have to supply one or more DefaultTopics that will handle all topics other than those that constitute a reply to the NPC's question (otherwise the Topic Entries available in the the NPC's current ActorState will also be available). Then we have to prevent Topic Entries from outside the ConvNode from being suggested. We may also want to prevent the player character from terminating the conversation (by saying goodbye, or walking way, or waiting for the NPC to become bored); and we may also want the NPC to keep nudging the player character for an answer if the player keeps entering non-conversational commands.

We'll illustrate all this with an example, and then explain the properties and methods involved:

```
+ ConvNode 'lighthouse'
   npcGreetingList: StopEventList
    {
     [
       'Seeing you approach, Bob turns to you and asks, <q>Hello,
        there! I hear you\'ve been asking about the lighthouse. Would you like
        me to take you there?</q> ',

       'Bob looks up at your approach and says, <q>Are, you\'re back. Shall we
        go to the lighthouse now? </q>'
    ]
   }

   npcContinueList: ShuffledEventList
   {
     [
        '<q>I asked you a question,</q> Bob reminds you, <q>Do you want me to
         show you the lighthouse?</q> ',

        '<q>I didn\'t think I\'d asked a particularly difficult question,</q>
         Bob remarks, <q>Do you want me to show you the lighthouse or don\'t
         you? A simple yes or no will do!</q> ',

        '<q>I\'m still waiting for your answer,</q> says Bob, <q>Do you want
         me to take you to the lighthouse, yes or no?</q> '
     ]
    eventPerCent = 67
   }
   limitSuggestions = true
   
   canEndConversation(actor, reason)
   {
      switch(reason)
      {
         endConvBye:
           "<q><q>Goodbye</q> isn't an answer,</q> Bob complains, <q>I asked
            if you wanted me to show you the lighthouse; do you?</q>";
            return blockEndConv;
            
         endConvTravel:
           "<q>Hey, don't walk away when I'm talking to you!</q> Bob complains,
            <q>I asked you a question! Do you want me to take you to the
            lighthouse?</q> ";
            return blockEndConv;
            
         default:
            return nil;
      }
   }
;


++ YesTopic, SuggestedYesTopic
   "<q>Yes, I would like you to show me the lighthouse,</q> you say.\b
    <q>Right; I can't take you there now. Come back and meet me here at six,</q>
    he tells you. "
;

++ NoTopic, SuggestedNoTopic
  "<q>No, I've been warned that the lighthouse is not a good place to visit,</q>
   you reply.\b
   <q>Very well,</q> he shrugs. "
;

++ DefaultAnyTopic, ShuffledEventList
   [
     '<q>Don\'t try to change the subject, I asked you if you want me to
      show you the lighthouse,</q> Bob replies, <q>So, do you?</q><.convstay> ',
      
     '<q>That doesn\'t answer my question,</q> he complains, <q>Do you want
      me to take you to the lighthouse?</q><.convstay> ',
      
     '<q>I asked you if you wanted me to show you the lighthouse,</q> he
      reminds you, <q>Do you?</q><.convstay> '
   ]
;
```

Note the use of the `<.convstay>` tags in the DefaultAnyTopic. By default any conversational command will take us out of the current ConvNode. The <.convstay> tag keeps the current ConvNode active, which is what we want all these default responses to do. The alternative would be to change the default behaviour of the ConvNode by changing its `isSticky` property to true; then a conversational command won't take us out of the ConvNode unless it contains an <.convnode> tag. We could then use <.convode node> to switch to another ConvNode or <.convnode nil> to leave the ConvNode without switching to another (not because nil has any special meaning in this context, but because there's unlikely to be a ConvNode called nil and thus, failing to find such a ConvNode, the ConvNode switching mechanism will switch to no node at all.

The methods and properties of ConvNode we might be most interested in making use of include:

- `isSticky` – if this is true then a conversational command won't cause the actor to switch out of this node unless the response includes an explicit <.convnode> tag. If it is nil we need to use explicit <.convstay> tags in responses to stay in this ConvNode.
- `name` – a single-quoted string that uniquely identifies this ConvNode among the ConvNodes for the same actor. We normally assign this property via the ConvNode template.
- `npcContinueList` – if supplied, this should be assigned an EventList object that displays messages in which the actor prompts the player character to respond to his/her question.
- `npcContinueMsg` – if supplied, this is an alternative to npcContinueList (we shouldn't define both properties); this would be a double-quoted string displaying a message the actor uses to prompt the player character to answer the question on each turn that the player issues a non-conversational command.
- `npcGreetingList` – if supplied, this should be assigned an EventList object that displays a series of messages representing the NPC initiating the conversation (generally by posing a question to which one of the TopicEntry responses defined in the ConvNode will be the answer).
- `npcGreetingMsg` – if supplies, a double-quoted string that displays the message representing the NPC initiating the conversation (generally by posing a question to which one of the TopicEntry responses defined in the ConvNode will be the answer). This is an alternative to npcGreetingList.
- `autoShowTopics()` - if true then a list of suggested topics will be shown on entering this node. Note that this method is already defined to show suggested topics if the ConvNode contains any SpecialTopics (which we'll explain below). This is a sensible default, but we may want to override it in particular circumstances.
- `limitSuggestions` – if this is set to true then only SuggestedTopics defined within the ConvNode will be listed in response to a **topics** command or to suggested topics being listed for any other reason. This should be used when we use DefaultTopics to trap any conversational commands not specifically handled by ConvNode, in which case Topic Entries outside the ConvNode (in the current ActorState or Actor) will not be reachable. We can also use this property

on ActorStates for a similar purpose (when SuggestedTopics defined on the Actor are made unreachable by the DefaultTopics in the ActorState).

- `canEndConversation(actor, reason)` – determines whether or not the player character is allowed to end the conversation when we're in this ConvNode. Return true to allow the conversation to end, or nil or `blockEndConv` to prevent it from ending. We use the last of these values to suppress displaying the npcContinueMsg or an item from the npcContinueList on the same turn; normally canEndConversation() should display a message possibly in the form of an objection from the NPC, explaining why the conversation cannot be ended; if we've already displayed such a message we don't also to see one of the NPC's nag messages appear on the same turn. The *reason* parameter is one of enums endConvBye, encConvTravel or endConvBoredom (the player has tried to say goodbye, the player has issued a movement command, or the NPC has exceeded its boredom threshold).
- `noteActive()` - called when the actor enters this ConvNode; by default this method schedules a listing of suggested topics if autoShowTopics() returns true.
- `noteLeaving()` - called when the actor is about to leave this ConvNode; by default this doesn't do anything.

For further details, look up ConvNode in the *Library Reference Manual*.

The examples we have seen so far have used YesTopic and NoTopic, but we can use any of the kinds of Topic Entry we like in a ConvNode. We can also a further kind of Topic Entry that's only available in a ConvNode, a `SpecialTopic`. A `SpecialTopic` in principle allows us to define just about any kind of conversational response we like in a ConvNode, allowing a player character to give rather more nuanced responses to what an NPC has just said than ask, tell, yes, or no normally allows. In practice there are of course a number of restrictions, but SpecialTopics certainly allow us greatly to extend the range of what the player character can reply.

A SpecialTopic can match the player's input in one of two ways:

1. Through its `keywordList` property; or
2. Through its `matchPat` property.

With the second of these, we can define `matchPat` as a regular expression which the player's input must match if this SpecialTopic is to match. This provides maximum flexibility, but may be more flexibility than we often need, especially if we're not that confident at writing regular expressions! In either case, we can use the first method instead (which is probably the method more commonly used when defining SpeciialTopics).

This first method is to define a list of keywords (as a list of single-quoted strings); the SpecialTopic will be matched if each word in the player's input matches one of the keywords. But if more than one SpecialTopic in the current ConvNode could match the player’s input, then all the matches will be ignored (in other words, the player’s input must uniquely identify one SpecialTopic for a match to take place).

For example, suppose we wanted one possible response in a ConvNode to be **tell the truth**. We could define the keywordList as ['tell', 'the', 'truth']. This would then be matched if the player typed **tell the truth** or **tell truth** or **truth** or perhaps even **tell** or **the** (if these words by themselves are not in the keywordList of any other SpecialTopic), but it would not match **tell him the truth** or **tell a truth** or **tell the truth reliably** or anything else that contains one or more words not in the keywordList.

Since a player can hardly be expected to guess the wording of a SpecialTopic, all SpecialTopics are automatically SuggestedTopics (in fact they’re SuggestedTopicTrees, so that the same suggestion will apply to any AltTopics they contain). This means that we must remember to define the `name` property for every SpecialTopic. In the previous example, we would define the name property as 'tell the truth'.

You might think that faced with a prompt that said something like:

```
You could lie, be evasive, or tell the truth
>
```

Players would realize that they’re mean to type one of these three options. Not so. You can guarantee that at least some players will try everything *except* the wording you’ve supplied in plain sight, and will then complain like mad that your game is buggy because it didn’t understand what they typed and they couldn’t guess what they were meant to type. There’s a limit to how far you can guard yourself against such pervasive player perversity, but you can guard yourself to some extent by thinking of some of the possible alternative phrasing people might try, such as **tell him the truth** or **tell bob the truth** or **be truthful** and add additional words to your keywordList accordingly (good beta-testers can also help with this problem, of course). Thus our sample SpecialTopic might look like this:

```
++ SpecialTopic
   name = 'tell the truth'
   keywordList = ['tell', 'him', 'bob', 'the', 'truth', 'be', 'truthful' ]
   topicResponse = "<q>Well, to be honest,</q> you say, <q>I've been having
    an affair with your wife for the last ten years.</q>\b
    <q>Thank goodness!</q> he declares, <q>Now I've got an excellent reason
    to divorce the wretched woman!</q>"
;
```

Once again, this can be defined more concisely using a template:

```
++ SpecialTopic 'tell the truth'
   ['tell', 'him', 'bob', 'the', 'truth', 'be', 'truthful' ]
    "<q>Well, to be honest,</q> you say, <q>I've been having
    an affair with your wife for the last ten years.</q>\b
    <q>Thank goodness!</q> he declares, <q>Now I've got an excellent reason
    to divorce the wretched woman!</q>"
;
```

Just to underline the point, SpecialTopics can *only* be used in ConvNodes. If you want something like SpecialTopic functionality elsewhere you could try downloading the SayQuery extensions from the IF-Archive.

Finally, ConvNodes themselves can be located either in the associated Actor or in one of that Actor's ActorStates. It's always okay to put a ConvNode directly in the Actor, and that's probably the best option.

14.8 NPC Agendas

Most of what we have seen so far has been about how we can make NPCs react to what the player character is doing. But our NPCs may feel more realistic if we can make them pursue their own agendas. We can do this with the **AgendaItem** class. By adding defining AgendaItems for our NPCs we can have them carry out certain actions as and when certain conditions become true, for example:

```
+ bobWanderAgenda: AgendaItem
      isReady = (bob.curState == bobWalking)
      initiallyActive = true
      agendaOrder = 10
      routeList = [highStreet, northStreet, southPark, northPark]
      invokeItem()
      {
         local idx = routeList.indexOf(bob.getOutermostRoom);
         if(idx && idx < routeList.length())
         {
            local dest = routeList[++idx];
            bob.scriptedTravelTo(dest);
            if(idx >= routeList.length())
              isDone = true;
         }
      }
;

+ bobAngryAgenda: AgendaItem
      isReady = (bob.canSee(mavis))
      invokeItem()
      {
         "<q>You hypocritical old woman!</q> Bob storms at Mavis, <q>You sit
           there like some stern old maiden aunt, but I know just what you
           were in your youth - you were a <i>trollope</i>! ";
         isDone = true;
      }
;
```

The first of these, **bobWanderAgenda**, is initiallyActive, so it will fire as soon as its **isReady** property becomes true (which is when Bob enters the **bobWalking** state). While this AgendaItem is ready, its **invokeItem()** method will be called each turn until its **isDone** property becomes true, whereupon the AgendaItem will be removed from Bob's **agendaList**. In this example the invokeItem() method makes Bob travel one step of the way through the route defined in the **routeList** property (a custom property we have defined specially for the purpose). The **bobAngryAgenda** is not initially active, however, so it won't do anything at all until we add it to Bob's **agendaList**, which we can do by calling **bob.addToAgenda(bobAngryAgenda)**. Once we've done this, bobAngryAgenda's **invokeItem()** method will be called as soon as Bob can see Mavis, with one proviso: only one AgendaItem can fire for a given actor on any one term, so if Bob should happen upon Mavis while bobWanderAgenda is talking Bob on his walk, Bob will simply walk on and ignore Mavis. The reason is that we have given bobWanderAgenda an **agendaOrder** of 10, much lower than the default value of 100, and that in cases where more than one AgendaItem is ready on a single term, the one with the lowest **agendaOrder** is the one that's used.

There's one further point to bear in mind here: bobAngryAgenda displays some text in its **invokeItem()** method, but this text won't actually be displayed on the screen unless the player character can see Bob at the time when this **invokeItem()** method is invoked (although everything in the method apart from the displaying of text will be carried out as usual). Normally, this is just what we want, since it means we don't have to worry about messages appearing which describe what invisible actors are doing off-stage. Every now and again, however, it can produce puzzling behaviour, especially if the NPC in question moves in or out of scope during the turn in question. Not only is output from the **invokeItem()** method suppressed when the corresponding actor is out of sight, but so is that from any method or function called by **invokeItem()**, and this can sometimes be even more puzzling when we don't get the output we're expecting.

If for any reason we're not seeing the output from an AgendaItem we want to see, this is very likely to be the reason. If we want to override the library's output suppression in such a case the way to do it is with the **callWithSenseContext()** function (for the details of which, see the *Library Reference Manual*). To force output of text to the screen no matter what the sense context, use nil for the first two arguments of **callWithSenseContext()** and an anonymous function to display the text as the third argument, for example:

```
callWithSenseContext(nil, nil, {: "This will display no matter what the
   player character can or cannot see. " });
```

To summarize: to make it possible for an AgendaItem to be activated it either needs to start out active (with **initiallyActive** set to true) or else be added to its actor's **agendaList** by calling its actor's **addToAgenda(item)** method (where *item* is the AgendaItem in question). AgendaItems should be located directly in the actor to which they relate. The most commonly important properties and methods of AgendaItem are:

- **agendaOrder** – the priority of this AgendaItem; the lower the agendaOrder, the higher the priority; the default value is 100.
- **initiallyActive** – set to true if this AgendaItem should be added to its actor's

agendaList at the start of the game.

- **isDone** – set this to true when the AgendaItem has finished doing whatever it needs to do; the AgendaItem will then be removed from its actor's agendaList.
- **isReady** – this should become true when we want the invokeItem() method to be called. Usually we define this as a method or expression that becomes true when the appropriate conditions obtain, but we could also set its value from an external method.
- **getActor()** - returns the actor with which this AgendaItem is associated.
- **invokeItem()** - this method should contain the code we want to execute once isReady becomes true.
- **resetItem()** - this method is automatically called when we add this item to its actor's agendaList; its function is to reset isDone to nil provided isDone is a simple true/nil value and not code that returns a value; the purpose is to allow us to reuse an AgendaItem without having to reset the isDone flag explicitly.

There are also two special kinds of AgendaItem we can use: **DelayedAgendaItem** and **ConvAgendaItem**. A **DelayedAgendaItem** is simply an AgendaItem that becomes ready so many turns in the future. We can add a DelayedAgendaItem to an actor's agendaList and set the delay at the same time by calling:

```
actor.addToAgenda(myDelayedAgendaItem.setDelay(n));
```

Where *n* is the number of turns in the future at which we want **myDelayedAgendaItem** to become ready. If we want to impose an additional condition, e.g. we want Bob to be able to see Mavis as well, we must combine this with **inherited**:

```
+ bobDelayedAngerAgenda: DelayedAgendaItem
    isReady = (inherited && bob.canSee(mavis))
    ...
;
```

A **ConvAgendaItem** is one that becomes ready when (a) the actor can speak to the player character and (b) the player character hasn't conversed with the actor this turn. We can use this to allow an actor to pursue his or her own conversational agenda once he or she gets a chance to get a word in edgeways. If we want the actor to try to converse with some other NPC, we can change the **otherActor** property of the ConvAgendaItem to that other NPC (it's the player character by default). If we want an AgendaItem to be both a ConvAgendaItem and a DelayedAgendaItem, we can just list both of these in the class list, e.g.:

```
+ bobDelayedAngerAgenda: DelayedAgendaItem, ConvAgendaItem
    isReady = (inherited && bob.canSee(mavis))
    ...
;
```

We'll say a bit more about ConvAgendaItem in the next section; in the meantime, for more information about AgendaItems look up AgendaItem in the *Library Reference Manual*.

14.9 Making NPCs Initiate Conversation

We've now met a number of ways in which we can make an NPC initiate conversation. We could just use a Fuse or Daemon and have it display the text of what we want the NPC to say. Or we can call the actor's `initiateConversation()` method from somewhere. Or we can use an AgendaItem, or better, a ConvAgendaItem. Or we can use a mechanism we've not yet met, namely calling the actor's `initiateTopic(obj)` method.

Often, the best place to start will be with a ConvAgendaItem, since this already checks that the NPC is in a position to converse with the player character and that they haven't already conversed on that turn. If we want to build in a delay we can simply add DelayedAgendaItem to the class list, as we've already seen. If we want to add further conditions to when the NPC should make his or her conversational gambit we can either do so by defining `isReady = (inherited && ourExtraConditions)` or by waiting until we're ready before adding the ConvAgendaItem to the NPC's agendaList. The question is then what to put in the ConvAgendaItem's `invokeItem()` method. There are basically two main options. The first is simply to display some text; the second is to use `initiateConversation()`.

The first is the one to use if we just want the NPC to make a remark which doesn't require any particular response on the part of the player character. For example:

```
+ bobLighthouseAgenda: ConvAgendaItem
    isReady = (inherited && bob.canSee(lighthouse))
    invokeItem()
    {
       "<q>Look! There's the lighthouse!</q> Bob declares. ";
       isDone = true
    }
;
```

This works best when we don't have to worry about switching the actor between a ConversationReadyState and InConversationState (in the above example Bob is more likely to be in an AccompanyingState, since that's the most likely way in which he and the player character will come upon the lighthouse together).

The main alternative is to call the actor's `initiateConversation()` method from a ConvAgendaItem. There may be a number of situations when we want to do this, in particular:

1. The NPC and the player character are not currently conversing, and we want the NPC to start a conversation by making some remark.

2. The NPC and the player character are not currently conversing, and we want the NPC to start a conversation by asking a question to which the player character either could or must reply.
3. The NPC and the player character are currently conversing, and we want the NPC to change the subject by asking a question to which the player character either could or must reply.

In case 1 we could simply call **initiateConversation()** with two nil arguments:

```
+ talkAgenda: ConvAgendaItem
    invokeItem()
    {
         getActor.initiateConversation(nil, nil);
         "<q>Blah, blah blah!</q>";
         isDone = true;
    }
;
```

When called with two nil arguments initiateConversation won't trigger a ConvNode, but it will cause a switch from a ConversationReadyState to an InConversationState, and it will register that a conversation between the player character and the NPC is now in progress; otherwise it will leave the NPC in the same state. This means that it will in fact work equally well whether or not the NPC and the player character were already conversing.

By the same logic, we scarcely need to distinguish between cases 2 and 3. We can either call **initiateConversation(nil, newNode)** or **initiateConversation(newState, newNode)**, depending on whether or not we want the NPC to switch to a specific ActorState. Either way we can rely on newNode's **npcGreetingMsg** or **npcGreetingList** to supply the NPC's question, so we can just write something like:

+ questionAgenda: ConvAgendaItem

```
    invokeItem()
    {
         getActor.initiateConversation(inquisitiveState, 'new-query');

         isDone = true;
    }
;
```

There's one further way we can make NPCs take some conversational initiative, and that's through **InitiateTopic** objects. An **InitiateTopic** is a kind of TopicEntry, and we can use it just like any other kind of TopicEntry. The difference is that an **InitiateTopic** is not triggered by any player command, but by the actor's **initiateTopic(obj)** method, which will trigger the **InitiateTopic** (if one exists) whose **matchObj** is *obj* (or contains *obj* in its **matchObj** list). For example, suppose we have an NPC in an AccompanyingState who comments on some of the locations as she visits them for the first time. We could implement this as follows:

```
+ sallyFollowing: AccompanyingState
    specialDesc = "Sally is at your side. "
    arrivingTurn() { sally.initiateTopic(sally.getOutermostRoom); }
;

++ InitiateTopic, EventList @forest
   [
      '<q>What a gloomy place!</q> Sally declares. '
   ]
;

++ InitiateTopic, EventList @meadow
   [
      '<q>Look at the tower, over there!</q> Sally tells you, pointing to
      the north. '
   ]
;
```

Here we've made the InitiateTopics EventLists as well so that Sally's comments will only be displayed the first time she enters each location alongside the player character.

14.10 Giving Orders to NPCs

In Interactive Fiction it's common for players to be able to give commands to NPCs, like **bob, go north** or **mavis, eat the cake**. By default TADS 3 will respond to all such commands with "Bob refuses your request" (or whichever NPC it is that refuses your request). We should now give a little thought to how we can change this, either to make an NPC obey a command, or to customize his or her refusal.

The method that determines whether an NPC will obey or refuse a command is `obeyCommand(issuingActor, action)`, where *action* is the action that *issuingActor* wants the NPC to perform. This is called on the NPC Actor object, which by default calls `obeyCommand(issuingActor, action)` on the Actor's current ActorState. By default the result is to return nil, which causes the command to be rejected. One way to make an actor obey a command is to override this method so that it returns true (at least for the action in question); to make an actor jump when we order him to:

```
modify ActorState
   obeyCommand(issuingActor, action)
   {
      if(action.ofKind(JumpAction))
        return true;
      return inherited(issuingActor, action);
   }
;
```

There's one global tweak we might want to make on the Actor class, regardless of what else we want to do with NPC commands. The standard library treats **mavis, save** or **bob, restart** or **sally, quit** just like any other commands targeted at NPCs, but **save**, **restart**, **quit** and a handful of other commands of this sort are really meta-commands, not commands relating to the game state, and it simply makes no sense to direct them to NPCs. In a game in which all commands targeted at NPCs are refused with the default message, this may not matter, but as soon as we start customizing responses or allowing some commands to be obeyed, we may want to nip these meta-commands in the bud, rejecting them all with a special message. Since all such meta-command actions will belong to the SystemAction class, which shouldn't contain any other kinds of action, this is relatively easy to trap. We can simply define this somewhere in our game:

```
modify Actor
     obeyCommand(issuingActor, action)
     {
        if(action.ofKind(SystemAction))
        {
           libMessages.systemActionToNPC();
           return nil;
        }

        return inherited(issuingActor, action);
     }
;
```

We could, of course, use a different message here if we wanted to. We can also customise the message an actor uses to refuse a standard command by customizing the Actor's `defaultCommandResponse(fromActor, topic)` method (where *topic* is the action the Actor is being asked to carry out); for example:

```
mavis: Person 'old aunt woman/mavis*women'  'Aunt Mavis'
   isHer = true
   isProperName = true
   defaultCommandResponse(fromActor, topic)
   {
      "<q>Don't you tell me what to do, young man!</q> she snaps. ";
   }
;
```

The `obeyCommand()` method on ActorState treats a command targeted at the NPC as a kind of conversational command. We can customize the NPC's reaction to commands by defining `CommandTopic` entries, which work just like other Topic Entries except that they match on action classes rather than game objects or topics. For example, we could customize Bob's response to **bob, jump** by defining a `CommandTopic` like this:

```
+ CommandTopic @JumpAction
   "<q>Jump!</q> you cry.\b
    <q>No - go jump yourself!</q> he replies. "
;
```

Alternatively, we could make Bob carry out the command by using a newActorAction() macro in the topicResponse():

```
+ CommandTopic @JumpAction
```

```
   topicResponse()
   {
      "<q>Jump!</q> you cry.\b";
      newActorAction(bob, Jump);
   }
;
```

We can also define a `DefaultCommandTopic` to catch all the commands we haven't written specific CommandTopics for; for example, as an alternative to overriding defaultCommandResponse() on Mavis we could give her a DefaultCommandTopic:

```
+ DefaultCommandTopic
   "<q>Don't you tell me what to do, young man!</q> she snaps. "
;
```

Note that we don't have to override `obeyCommand()` to use CommandTopics. The full default behaviour of `ActorState.obeyCommand()` is to handle the order as a conversational command (handled by CommandTopics) and then return nil. Conversely defining CommandTopics has nothing to do with whether or not the command is obeyed (unless we explicitly make the actor carry out an action with `newActorAction`, as in the above example).

There is one rather obvious limitation to this mechanism: a CommandTopic matches on the Action class, but not on any of the objects involved in the command. It doesn't provide a ready mechanism for handling **mavis, eat cake** differently from **mavis, eat table** or **mavis, eat your hat**, let alone distinguishing **bob, put the red ball in the brown box** from **bob, put the flaming torch in the vat of petrol**. If you want to do be able to do this, then probably the easiest way is to use the TCommand extension, which can be found in the ../lib/extensions directory of your TADS 3 installation. This would allow you to define TCommandTopic objects like:

```
+ TCommandTopic @PutInAction
    "<q>Put the red ball in the brown box, would you?</q>\b
     <q>Okay,</q> Bob agrees. "
     matchDobj = redBall
     matchIobj = brownBox
     obeyCommand = true
;
```

The alternative, if you need to get at the actions involved in the command, is to inspect `action.getResolvedDobjList()` and/or `action.getResolvedIobjList()` (where *action* is the action the NPC has been ordered to carry out); this is perfectly doable, but you may end up reinventing the TCommand wheel.

14.11 NPC Travel

There may be some NPCs who have a good reason for staying right where they are throughout the course of a our game, but the chances are that at least some of our NPC will need to move around. We've already seen how we can use an

AccompanyingState or a GuidedTourState to make an NPC move around with the player character, but we may just as likely want some of our NPCs to travel around independently of the player character.

The first golden rule of NPC Travel is *never* move an Actor (whether an NPC or the player character) with **`moveInto()`**. The problem is that the library tries to calculate a containment path between the actor's current location and the new location, and then move via the actor via that path, which in the general case can all too easily result in a run-time error or some other undesired effect. If we want to move an actor around by authorial *fiat* that magically transports him, her, or it across the map we should use **`moveIntoForTravel(dest)`** instead (where *dest* is the destination we want to move the actor to).

This may not always be our best option, however, since we may more normally want our NPCs to move around the map in much the same way as the player character does, rather than having them suddenly vanish from one location and appear in another. In that case it's probably easiest to use either **`scriptedTravelTo(dest)`** or make the NPC carry out a TravelVia action on the appropriate TravelConnector (these are in fact more or less two different ways of doing the same thing).

The **`scriptedTravelTo(dest)`** method allows us to make an actor travel to a neighbouring location, so, for example, if Bob is in the east end of the hall and we want him to move to the west end of the hall, we could call:

```
bob.scriptedTravelTo(hallWest);
```

Note, however, that this will only work if Bob starts in a location that's directly adjacent to the destination to which we want him to move; if he's not, the method will simply do nothing. If we need an NPC to find a path over a longer distance, we should probably use the pathfind.t extension in the ../lib/extensions folder.

We could **`scriptedTravelTo()`** in an AgendaItem to make an actor travel along a pre-computed path, for example:

```
class TravelAgendaItem: AgendaItem
    travelPath = nil
    invokeItem()
    {
       local actor = getActor;
       local path = nilToList(travelPath);
       local loc = actor.getOutermostRoom();
       local idx = path.indexOf(loc);
       if(idx && idx < path.length())
          actor.scriptedTravelTo(path[++idx]);

       if(idx == nil || idx >= path.length())
         isDone = true;
    }
;
```

For particular instances we could then define travelPath to be a list of adjacent rooms starting with the initial location and ending with the destination; an NPC following this TravelAgendaItem should then proceed one step of the journey each turn.

An alternative is to use `newActorAction()` (or `nestedActorAction()`) to make an actor carry out a travel command. For example, if we want to send Bob through the red door, we could write:

```
newActorAction(bob, TravelVia, redDoor);
```

Or, if we simply want to send Bob off to the east, we could use:

```
newActorAction(bob, East);
```

The final way to make an actor move around is with its `travelTo(dest, connector, backConnector)` method. Here *dest* is the destination to which we want to send the actor, *connector* is the TravelConnector the actor should travel via to get there, and *backConnector* is the connector leading back from the the destination to where the actor is coming from (this is used to describe the actor's arrival). If we don't want to specify the backConnector ourselves, we can get it from:

```
connector.connectorBack(actor.getTraveler(connector), dest)
```

The fact that `travelTo()` requires these three parameters may make it the least convenient method to use. Unlike the last few methods we've been looking at it doesn't carry out preconditions of travel such as opening closed doors; this is fine if the actor is meant to be a ghost who can walk through locked doors, but may not always be ideal in other circumstances. For further details of this and other ways of moving actors around, see the "NPC Travel" article in the *TADS 3 Technical Manual*.

In addition to be able to move NPCs around, we may also like to have some control over how that travel is described. The library provides a number of standard messages for describing NPCs arriving and departing within sight of the player character, and as plain vanilla descriptions of NPCs moving around they do well enough. On occasion, however, we may want something more colourful: "Bob stomps off to the west", say, rather than just "Bob leaves to the west."

To achieve this, we can override one or more of the following methods on the NPC's ActorState:

- `sayDeparting(conn)` – generic message for departing via a TravelConnector
- `sayDepartingDir(dir, conn)` – directional departure via a RoomConnector
- `sayDepartingThroughPassage(conn)` – departure via a ThroughPassage or Door
- `sayDepartingViaPath(conn)` – departure via a PathPassage
- `sayDepartingUpStairs(conn)` – departure up a StairwayUp

- `sayDepartingDownStairs(conn)` – departure via a StairwayDown

There's also a corresponding set of `sayArrivingXXX()` methods to describe an actor arriving via these various kinds of TravelConnector. Which of these methods is used to generate the message depends on what kind of TravelConnector the NPC is travelling via. For example, to customize the messages to use when Bob departs in a particular direction or down a flight of stairs we could write:

```
+ bobWalking: ActorState
  sayDepartingDir(dir, conn)
  {
     "Bob stomps off to the <<dir.name>>. ";
  }
  sayDepartingDownStairs(conn)
  {
     "Bob runs lightly down <<conn.theName>>. ";
  }
;
```

For a fuller list (and account) of these methods, look up TravelMessageHandler in the *Library Reference Manual.* You should also look up AccompanyingInTravelState where all these methods are overridden to use `sayDeparting(conn)`, the reason being that this is the message that's used when the NPC is in the process of accompanying the player character on his/her travels.

14.12 Afterword

As you'll have gathered if you've read this far, TADS 3 provides a rich set of tools for programming NPCs, but it may all seem a little overwhelming at first. Once you start coding your own NPCs it should all start to fall into place, and you'll soon start to pick up what class or method you need to use for which job. If in the meantime you need more guidance on NPCs, there are several places you can look:

1. If you haven't already done so, you should read the articles on "Creating Dynamic Characters", "Choosing a Conversation System" and "Programming Conversations with NPCs" in the *TADS 3 Technical Manual*

2. You may well find it helpful to download and use the "TADS 3 Dynamic Characters Quick Reference"; this attempts to summarize much of the information in this chapter in a chart that can be printed out on two sides of a (letter-size or A4) page; it can be obtained from http://users.ox.ac.uk/~manc0049/TADSGuide/QRefs.zip.

3. You can browse the *TADS 3 Reference Manual* for the various classes introduced in this chapter.

4. You can look at Chapter 4 of *Getting Started in TADS 3* and follow the way the two NPCs are developed in that and subsequent chapters.

5. You can attempt the following exercise and then compare your attempt with the

sample game lighthouse.t

Exercise 20: Both the NPC articles in the *TADS 3 Technical Manual* and several of those in this chapter refer to a character called Bob who stacks cans and mutters darkly about a lighthouse and some "troubles", so try writing a small game based on that. Here's some further suggestions: The player character is new to the town and has just gone into Bob's shop, where Bob is busily stacking cans and a blonde woman (let's call her Sally) is inspecting the clothes on the clothes rack. Sally is too interested in the clothes to engage a stranger in conversation, but Bob is more talkative. You can ask him about a number of topics, but if you ask him about the town, he'll mention the lighthouse and the troubles, and then clam up on those topics. When Sally hears Bob mention the troubles she goes outside. When the player character leaves the shop Sally comes up to him and asks whether he wants her to show him the lighthouse. The player can reply yes or no or ask why she's offering. If the player replies yes, Sally leads the player character to the lighthouse and then invites him to lead the way inside. On the lowest floor of the lighthouse there's nothing but an abandoned storeroom and an abandoned office, which Sally comments on the first time she follows the player character there. Halfway up a spiral staircase is an oak door. When the player character goes through the door he meets more than he bargained for.

When you've got as far as this as you want to, compare your version with the lighthouse.t sample game.

15 MultiLocs and Collectives

15.1 MultiLocs

At a stretch we could regard MultiLocs and Collectives as complementary: MultiLocs cater for one object being in several places at once, while Collectives (and CollectiveGroups) help deal with multiple objects in the same place. We'll start with MultiLocs.

Generally, the whereabouts of an object in a TADS 3 game is defined by its `location` property, which would suggest that, like most objects in the real world, an object can only be in one place at a time. But there are three situations where we really do want a game object to be in several places at once:

1. The object in question straddles the border of several rooms, such as a fountain that stands at the centre of a square we've implemented as four different rooms.
2. The object is a distant object (like the sun, the moon, or a faraway mountain) that's visible from many different locations.
3. The object is some kind of `SenseConnector` that needs to be in several locations at once in order to provide a sensory link between those locations. We'll look at this use of `MultiLoc` more in the next chapter.

For these three situations we can use the `MultiLoc` class. `MultiLoc` is a mix-in class which generally needs to precede one or more Thing-derived classes in any class list. There are a number of ways in which we can specify which locations a MultiLoc object is present in:

1. We can simply list its locations in its `locationList` property; so, for example, for the fountain at the centre of a square we might define `locationList = [squareNE, squareNW, squareSE, squareSW]`.
2. We can define its `initialLocationClass` to be a class of object every member of which contains the MultiLoc in question. For example, if we were implementing an object to represent the sun we'd probably define `initialLocationClass = OutdoorRoom`. We can refine this further, if we wish, by overriding the `isInitiallyIn(obj)` method; for example if we wanted the sun to appear in every OutdoorRoom *except* those of the our ForestRoom class (where the leafy canopy obscures the sun) we could additionally define `isInitiallyIn(obj) { return !obj.ofKind(ForestRoom); }`.
3. We can simply define `isInitiallyIn(obj)` without defining `initialLocationClass`; then our isInitiallyIn() method will select locations

from every single object defined in the game, not just those objects of the **initialLocationClass** class.

4. If none of these methods provided sufficient flexibility, we could override **buildLocationList()** to return the list of locations we want (though it's hard to think of a case where we'd need to do this).

The two examples we've used above might look like this:

```
sun: MultiLoc, Distant 'bright sun' 'sun'
   "It's too bright to look at for long. "
   initialLocationClass = OutdoorRoom
   isInitiallyIn(obj) { return !obj.ofKind(ForestRoom); }
;

fountain: MultiLoc, Container, Fixture 'ornamental fountain*fountains''fountain'
   "It's in the form of some improbable mythical beast gushing water out
    of an unmentionable orifice. "
   locationList = [squareNE, squareNW, squareSE, squareSW]
;
```

We've made the fountain a Container because we're envisaging it as the kind of fountain people could throw coins into. Note that if the player character were to throw a coin into the fountain, s/he'd be able to retrieve it equally well from any of the four corners of the square, since the fountain is in all of them.

Note also that although in this example (and in many common uses of MultiLoc) the locations the MultiLoc is present in are all Rooms, there's no rule restricting us to using Rooms; it's legal to use *any* Thing-derived object as a MultiLoc location (as it is for the location of any Thing). For example, if our MultiLoc were a SenseConnector establishing an aural link between two mobile phones, we could usefully locate it in the two mobile phones.

Note that while we can use the **isIn(loc)**, **isDirectlyIn(loc)** and **isOrIsIn(loc)** methods to test whether a MultiLoc is in *loc* (these methods are overridden on BaseMultiLoc to give sensible results), we can't determine the location of a MultiLoc by inspecting its **location** property, which will simply be nil. Normally this isn't an issue, but it could become an issue if we're writing a routine that's iterating over a group of objects, some of which are MultiLocs, and the routine assumes that all the objects have a valid **location**. If this is likely to be an issue in our game, one workaround is to define a location method on MultiLoc (or perhaps BaseMultiLoc) that returns a usable value, e.g.:

```
modify MultiLoc
   location()
   {
     /* if our locationList is empty, we don't have a location */
     if(locationList.length() < 1)
       return nil;
     /* if the player character is in one of our locations, we'll return that */
     if(gPlayerChar.isIn(self))
       return locationList.indexWhich({x: gPlayerChar.isIn(x) });

     /* otherwise return the first object in our locationList */
     return locationList[1];
   }
;
```

We can't just return `gPlayerChar.location` in the second case, because, for example, the player character might be on chair in a room that's one of the MultiLocs locations, and we want to report the MultiLoc's location as the room rather than the chair in this case. In any case, particular games might need to use some rather different algorithm to return a usable location for a MultiLoc, where it's necessary to do so; see, for example, the definition of `getDropDestination(obj, path)` on MultiLoc, which has to deal with an analogous problem (you may have noticed that this is one of several TADS library methods that take a 'path' parameter which we usually ignore; MultiLoc objects provide a situation where the 'path' parameter can actually be useful, to help determine where the player is accessing the object from).

We can use `moveInto(loc)` or `baseMoveInto(loc)` to move a MultiLoc, but the effect will be to move it out of all its existing locations leaving it only in *loc*. In some cases this may be just what we want, of course; for example, at sunset we could call `sun.moveInto(nil)` to remove the sun from everywhere at once. On other occasions we might want to be more selective what we're moving a MultiLoc in and out of, in which case we can use:

- `moveIntoAdd(loc)` – make the MultiLoc present in *loc* without removing it from any of its existing locations.
- `moveOutOf(loc)` – remove the MultiLoc from *loc* (without affecting the MultiLoc's presence anywhere else).
- `saveLocation()` - save the MultiLoc's locationList for future restoration; for example, we might call this at sunset before moving the sun into nil, so that we can restore the sun to the sky at sunrise. This method returns the list that's been saved, and we'd need to store this return value in an object property to be able to use it later.
- `restoreLocation(oldLoc)` – restore the MultiLoc's locationList to the value stored in *oldLoc*.
- `reInitializeLocation()` - rebuild the locationList according to the rules defined in initialLocationClass, isInitiallyIn() and/or buildLocationList().

One kind of situation MultiLoc is *not* designed for is representing multiple similar objects that occur in different locations, for example trees in a forest. For this kind of situation we should instead use `MultiInstance`. To use this class we set up a list of locations for the `MultiInstance` object in any of the same ways we would use for a

MultiLoc, but then define the actual object we want to appear in all those locations on the `instanceObject` property of the `MultiInstance` object. For example, to populate a forest with trees we could do this:

```
forestTrees: MultiInstance
   initialLocationClass = ForestRoom
   instanceObject: Decoration
   {
      'tall tree/trees' 'trees'
      "Tall trees surround you on every side. "
      isPlural = true
   }
;
```

The library will then put a copy of the instanceObject into each ForestRoom. Each of these copies will be an anonymous object, but we can obtain a reference to the instanceObject in any particular location *loc* with `forestTrees.getInstanceIn(loc)`. We can also move instances around with the `moveInto(loc)`, `moveIntoAdd(loc)`, and `moveOutOf(loc)` in the same way as for MultiLoc, except that in this case we're actually adding and removing instances, not the MultiInstance object. For example, if a group of loggers got to work in `forestEast` we could call `forestTrees.moveOutOf(forestEast)`.

A variation on multiple similar objects which exist in multiple locations is an object (such a river) which extends through several locations. For this we should use the `MultiFaceted` class, which works much the same as the MultiInstance class (from which it inherits); e.g.

```
river: MultiFaceted
    locationList = [riverBank, meadow, forestSouth]
    instanceObject: Fixture
    {
       'broad sluggish river*rivers' 'river'
       "The broad river flows sluggishly by. "
    }
;
```

The principal (and in effect, sole) difference between a MultiInstance and a MultiFaceted object is the way the latter handles pronouns. If a player types the command **x river** while the player character is in riverBank, then moves the player character to the meadow, then types **x it** the parser will recognize that 'it' is meant to refer to the river, and will take it to mean the river object that's in the meadow. The same would not happen with the trees in the forest, since they are MultiInstance trees, not MultiFaceted trees; the trees in the forest are numerically distinct (though similar); the river is numerically the same river in all its locations.

At first sight it may appear as if the MultiFaceted river should qualify for being a MultiLoc, but there is a difference. We *could* use MultiLoc for a river flowing *between* two locations (say the north bank and the south bank of the same segment of river), but we should not do so for a river *extending through* a number of locations. This may become clearer if we spell out the practical differences. Suppose that all the locations were in darkness: if we made the river a MultiLoc then a light shining on the river from any of its locations would illuminate all of them (but this would not happen with a MultiFaceted river). Similarly, suppose we made the river a RestrictedSurface (which could have things like a swan and a boat on it). If we made the river a MultiLoc then any swan or boat placed on it would be equally visible and accessible from every location through which the river passed; if that's not what we want then we need to make the river a MultiFaceted (for which every segment of the river is a separate object, even though all the segments are recognized as belonging to the same river).

15.2 Collectives

Dealing with related objects that occur in multiple places is one issue; the converse issue is dealing with related objects that are all in the same place, that is, objects that for some purposes we may prefer to treat together as a conglomerate while for others we need to treat individually. The `Collective` class is used for items that we want to deal with collectively when referred to in the plural, but individually when referred to in the singular. Collective is a mix-in class that is generally mixed in with some Thing-derived object; it's normally a physical object that we take a command to refer to when the objects it collects are referred to by the player in the plural. For example, the Collective might be a bag of marbles, book of matches, or bunch of grapes; **take marbles** would result in the bag of marbles being taken, while **take marble** would result in the taking of one individual marble.

To set up a Collective we must make it share the plural vocabulary of the objects it represents, and we needs its `isCollectiveFor(obj)` method to return true for all those objects. For example to set up a bunch of grapes we might start with the following:

```
class Grape: Dispensable, Food 'grape*grapes fruit' 'grape'
    isEquivalent = true
;
+ bunch: Collective, Dispenser 'bunch/grapes*grapes' 'bunch of grapes'
    "It's a big bunch of grapes. "
    myItemClass = Grape
    isCollectiveFor(obj) { return obj.ofKind(Grape); }
    cannotEatMsg = 'Don\'t be greedy! Eat the grapes one at time. '
    contentsListedInExamine = nil
;

++ Grape;
++ Grape;
++ Grape;
++ Grape;
++ Grape;
++ Grape;
++ Grape;
```

We'd probably want to tweak the desc property here, to vary the description a little according to the number of grapes left. When there's a lot of grapes on the bunch we probably don't want either a room description or an examine command to report the number of grapes in the bunch, hence `contentsListedInExamine = nil`. Once there's only a handful of grapes left in the bunch, we might want that to change; so perhaps a more sophisticated implementation would have `contentsListedInExamine = (contents.length < 5)` or some such. The vocabWords with the repeated 'grapes' may look odd, but the first 'grapes' is necessary to make the bunch match 'bunch of grapes' and the second is needed to make it act as a Collective for the individual grapes. The bunch of grapes is also a good candidate for being a Dispensable (we can't return the grapes to the bunch after taking them), so we define it as such.

15.3 CollectiveGroups

A `CollectiveGroup` is a little like a Collective, in that it's a single object representing a group of objects, but it differs from a Collective in two important ways: (a) it doesn't represent a physical object in its own right; and (b) we can be selective about what actions it handles (by default a CollectiveGroup handles only the **examine** command, leaving all other commands to be handled by the individual objects). A CollectiveGroup is thus an *abstract* object that can stand in to handle certain commands directed at a group of objects with the same plural noun. This is perhaps best clarified by means of an example:

```
bedroom: Room 'Bedroom'
   "This bedroom is quite cramped, but there's room for a bed, a chair, and
    a small desk. "
;

+ bed: Bed, Heavy 'narrow bed*furniture beds' 'bed'
  "It's narrow, but at least it's freshly made. "
  collectiveGroups = [furnitureGroup]
;

+ chair: Chair, Heavy 'hard wooden chair*furniture chairs' 'chair'
  "It's a hard wooden chair, solid but not particularly comfortable. "
  collectiveGroups = [furnitureGroup]
;

+ desk: Surface, Heavy 'small desk*furniture desks' 'desk'
  "It's just about large enough to write on. "
  collectiveGroups = [furnitureGroup]
;

+ furnitureGroup: CollectiveGroup, Heavy '*furniture' 'furniture'
   "There's a narrow bed, a hard wooden chair, and a small desk just
    large enough to write on. "
;
```

Without the CollectiveGroup, **x furniture** would list the bed, the chair, and desk and give each of their individual descriptions, but with the CollectiveGroup defined as above we get:

>x furniture
There's a narrow bed, a hard wooden chair, and a small desk just large enough to write on.

Note how this works: the `furnitureGroup` object shares the plural vocabulary 'furniture' (grammatically speaking one might call it 'collective', but it's plural from the point of view of TADS 3). To define which items belong to the furnitureGroup CollectiveGroup, we define the `collectiveGroups` property as `[furnitureGroup].` This is a list property so that the same item can belong to more than one CollectiveGroup (there is also a `collectiveGroup` property, but its use is deprecated). We add a NonPortable class to the CollectiveGroup's class list just to make sure the player doesn't walk off with the CollectiveGroup (which shouldn't be possible, but might become so under certain circumstances). We also give the CollectiveGroup a `desc` property that will be used instead of the `desc` properties of the individual member items when they're examined collectively.

We could also override the CollectiveGroup's `isCollectiveAction(action, whichObj)` method to allow the CollectiveGroup to respond to actions other than, or in addition to, **examine**. Here *action* is the action in question, *whichObj* is either DirectObject or IndirectObject (this would allow us to make the CollectiveGroup handle, say **throw ball at furniture** but not **throw furniture at ball**), and the method should return true if the CollectiveGroup is to handle the action. So, for example, to allow the furnitureGroup to response to **take**, **move**, **push** and **pull** as well as **examine** we could define:

```
+ furnitureGroup: CollectiveGroup, CustomFixture '*furniture' 'furniture'
   "There's a narrow bed, a hard wooden chair, and a small desk just
    large enough to write on. "
    isCollectiveAction(action, whichObj)
    {
        return action.baseActionClass
            is in (ExamineAction, TakeAction, PushAction, PullAction,
                     MoveAction);
    }
    cannotTakeMsg = 'The furniture is all too heavy to start moving around. '
;
```

In this example, all the furniture is fixed in place in the bedroom, and the CollectiveGroup is implemented as a further NonPortable object in the same location. This is the simplest kind of CollectiveGroup to implement. It's also possible to use a CollectiveGroup with a bunch of portable objects; the CollectiveGroup is then given no location at all, but is brought into scope whenever at least one of its member objects is in scope. This case is much harder to deal with, since whatever we want such a CollectiveGroup to handle, we have to know which of its members are in scope when that action is attempted and tailor the action accordingly. For more details see the notes on CollectiveGroup in the *TADS 3 Library Reference Manual*. For an example of a portable CollectiveGroup see the section on CollectiveGroup (mobile) in the *TADS 3 Tour Guide*.

The TADS 3 library does provide one kind of portable CollectiveGroup that's fairly easy to use, the `ItemizingCollectiveGroup`. When this kind of CollectiveGroup is examined, it *lists* such of its members as are currently visible, instead of describing them. It also lists items carried by the player character separately. For example, suppose we defined the following:

```
+ Thing 'blue marble*marbles' 'blue marble'
    collectiveGroups = [marbleGroup]
;

+ Thing 'red marble*marbles' 'red marble'
    collectiveGroups = [marbleGroup]
;

+ Thing 'green marble*marbles' 'green marble'
    collectiveGroups = [marbleGroup]
;

marbleGroup: ItemizingCollectiveGroup '*marbles' 'marbles'
;
```

We could then get the following transcript:

>x marbles
You see a blue marble, a red marble, and a green marble here.

>take red
Taken.

>x marbles
You see a blue marble and a green marble here. You are carrying a red marble.

16 Senses and Sensory Connections

16.1 The Five Senses

TADS 3 comes with handling for five senses: sight, sound, smell, touch and taste. The most elaborate provision is made for sight; fairly elaborate provision is made for sound and smell; and some basic provision is made for touch and taste. This largely reflects the way these senses are used in real life.

TADS 3 in fact defines a **Sense** class; **sight**, **sound**, **smell** and **touch** are all objects of this class. This scheme makes it possible to define additional senses if our game needs them, though doing so is beyond the scope of this manual (if this is something you need to do, look up Sense in the *Library Reference Manual* then study the definition of Sense in the related source code, along with the definition of the existing senses and the comments in the source code explaining how to set about creating new senses).

Although there's no taste sense object defined, handling taste and touch are fairly similar. To customize the responses to **taste something** and **feel something** we just override an object's **tasteDesc** and **feelDesc** properties, for example:

```
apple: Food  'round green red sweet firm juicy apple*apples fruit'  'apple'
   "It's round, with patches of both red and green. "
   feelDesc = "It feels reassuringly firm. "
   tasteDesc = "It tastes sweet and juicy. "
;
```

We *can* handle sound and smell in a similar fashion; for example:

```
apple: Food  'round green red sweet firm juicy apple*apples fruit'  'apple'
   "It's round, with patches of both red and green. "
   feelDesc = "It feels reassuringly firm. "
   tasteDesc = "It tastes sweet at juicy. "
   soundDesc = "The apple is obstinately silent. "
   smellDesc = "There's the faintest sweet apple smell. "
;
```

Normally, however, we'd use Noise and Odor to model sounds and smells (we'll meet these shortly below). In cases like the apple example above, though, using soundDesc and smellDesc is probably the way to go. Where these are simply given messages saying that we either sense nothing or very little, using a Noise or Odor object would be overkill.

Internally in the library touch is a little more complex than taste, since it is assumed that anything that can be touched can be tasted; the calculations the library needs to perform, then, are on whether a given object can be touched, or whether (for example) it's too far away or whether there's something obstructing access to it.

The remainder of this chapter will be largely concerned with sight, smell and sound.

16.2 Vaporous and Intangible

Most of the Things we have seen implemented so far in TADS 3 are solid objects like tables, chairs and rubber balls, but not everything we encounter in the world is of this type: for example smells and sounds, or smoke and light.

In TADS 3 the base class for things of this kind is **`Intangible`**. Apart from one or two specialized uses we shall encounter below, Intangible (as opposed to one of its subclasses) is not of much use, since it has no sensory presence at all.

If you look at the definition of Intangible in the *Library Reference Manual* you'll see that it has four properties (originally defined on Thing), which it overrides to be nil: **`sightPresence`**, **`soundPresence`**, **`smellPresence`**, and **`touchPresence`**. It also defines a catch-all **`dobjFor(Default)`** and **`iobjFor(Default)`** to trap all actions whatsoever and just display a **`notWithIntangibleMsg`**. This means that an object of class Intangible will not respond to any action, and has no sensory presence. This is why it is of limited use unless modified in some way, either in its subclasses (which between them cover most of the useful cases) or on a particular Intangible object.

One possibility that the library doesn't explicitly cater for is an object that can be touched, but not seen, heard or smelled, so we might just about want to define something like this:

```
+ Intangible 'force field*fields' 'force field'
    touchPresence = true
    sightPresence = true
    feelDesc = "It feels strangely solid for something you can neither see, hear
        nor smell. "

    dobjFor(Feel)  {  verify() {}  }
;
```

But this kind of thing is likely to be very rare.

A much more commonly useful type of Intangible is provided by the **`Vaporous`** class. This has a **`sightPresence`** of true and can be examined, listened to, looked behind, looked in, looked through, looked under, searched, or smelled. Looking in, under, through or behind a Vaporous will result in the display of the library's **`lookInVaporous`** message (unless we override the **`lookInDesc`** property). The Vaporous class is used for objects that can be seen but not physically manipulated, such as smoke, fog, fire, and beams of light. It might be smelly (like smoke) or noisy (like a crackling fire) but it can't be touched. If we want to change the message that's displayed for all the actions a Vaporous object won't handle we can override its **`notWithIntangibleMsg`**.

So, for example, we might define a beam of light like this:

```
+ Vaporous 'beam/light/dust/motes'  'beam of light'
   "The beam of light streams in through the window and plays upon the floor. "
   lookInDesc = "Motes of dust hang in the beam. "
   notWithIntangibleMsg = 'The beam of light is too insubstantial for that. '
;
```

16.3 Sensory Emanations

As intimated above, TADS 3 has a special way of dealing with sounds and smells. Although we *can* just assign a `smellDesc` and/or a `soundDesc` to various objects, the library encourages us to treat sounds and smells as objects in their own right. These are, of course, Intangible objects, and more specifically, objects of the `SensoryEmanation` class, or rather, of various classes that descend from `SensoryEmanation` (namely `Noise`, `Odor`, `SimpleNoise` and `SimpleOdor`).

These SensoryEmanation objects are generally located in the object to which they relate. For example, if we were implementing a smelly piece of cheese we might locate an Odor object in the cheese object; it were implementing a ticking clock we might locate a Noise object in the clock object. Commands like **smell cheese** and **listen to clock** would then be redirected to the Odor or Noise object located in the cheese or the clock.

The first two kinds of SensoryEmanation (Noise and Odor) can be a little complex to set up, since they have a whole host of properties describing the sound or smell under different conditions. In the list below we'll describe them in relation to sound, but exactly the same principles apply to smell. All these properties should be defined as double-quoted strings (or as methods that display some text):

- `sourceDesc` – the description of the sound appended to the source of this sound when that source is examined. For example, we might define this as "It's ticking. " so that "It's ticking. " would be appended to the description we got in response to **x clock**.
- `descWithSource` – the description of the *sound* when the player character can see its source. For example "A steady ticking comes from the clock. " in response to **listen to ticking**.
- `descWithoutSource` – the description of the *sound* when the player character can't see the source (maybe the clock is in a drawer or under a pillow, but the drawer or the pillow lets sound through). This might be something like "The ticking sounds steady but muffled. " shown in response to **listen to ticking**.
- `hereWithSource` – the description of the sound as shown in a room description or a response to an intransitive **listen** command when the player character can see the source of the sound. For example, "The clock ticks steadily. "
- `hereWithoutSource` – the description of the sound as shown in a room description or a response to an intransitive **listen** command when the source of the sound is hidden. For example, "There is a steady ticking sound. "

The xxxxWithoutSource properties come about usually because the source of the sound (or smell) is in a closed container made of some material that's opaque to sight, but transparent to sound or smell (see section 5.3.2 above).

The SensoryEmanation class has a number of other properties to help customize the way sounds and smells are displayed:

- `displaySchedule` – a list of numbers defining a series of intervals between one display of the hereWithXXXX message and the next. Once we reach the last number in the list that interval is repeated indefinitely, unless the last entry is nil, in which case the hereWithXXXX message is no longer displayed. This allows us to reduce the frequency with which we call the player's attention to an ongoing sound or smell. For example, if we defined displaySchedule = [2, 4, 8] on the clock, the "The clock ticks steadily" message would be repeated after two turns, then after a further four turns, then on every eighth turn after that.
- `isAmbient` – if this true/nil flag is true then this is a background noise or smell that won't be mentioned except in response to an explicit **listen** or **smell** command.
- `isEmanating` – this can be used as an on/off switch to define whether or not this SensoryEmanation is currently active; for example if the battery was removed from the clock so it stopped ticking, we could set isEmanating to nil on the ticking sound to suppress reports of the ticking.
- `noLongerHere` – a message to display when the player character can no longer hear/smell this SensoryEmanation, e.g. "You can no longer hear the ticking." The default is to display nothing.
- `displayCount` – the number of times in a row that a message has been displayed about this sound/smell. This is reset to 0 each time the SensoryEmanation comes into scope having been out of scope, so we could use it to vary the message we display about a sound or smell according to how many times it's already been mentioned.

This may all look quite complicated, but the purpose of using a SensoryEmanation rather than just defining a soundDesc or smellDesc on the object in question is that it gives us a far richer set of behaviour. By using a Noise or an Odor instead we can:

- get a note of the sound or smell added to the description of its source object when the source object is examined
- have the sound or smell reported in response to an intransitive **listen** or **smell** command
- have the sound or smell reported as part of the room description, and then mentioned again at author-defined intervals
- allows the player to listen to or smell the sound or smell, as well as its source; e.g. **listen to ticking** as well as **listen to clock**

The following example shows how we might use a Sound to define the ticking of a clock:

```
bedroom: Room 'Bedroom'
   "It's a small bedroom, with a small cabinet next to the bed. "
;

+ ComplexContainer, Heavy 'small cabinet*cabinets'  'small cabinet'
  subSurface: ComplexComponent, Surface { }
  subContainer: ComplexComponent, OpenableContainer { bulkCapacity = 7 }
;

++ clock: OpenableContainer, RestrictedContainer
   'black plastic alarm clock' 'black clock'
   "It's an alarm clock made of black plastic. "
   subLocation = &subContainer
   validContents = [battery]
;

+++ battery: Thing 'small aa battery*batteries' 'AA battery'
;

+++ Noise 'steady ticking sound/noise*sounds' 'ticking sound'
  sourceDesc = "It's ticking. "
  descWithSource = "A steady ticking comes from the clock. "
  descWithoutSource = "The ticking sounds steady but muffled. "
  hereWithSoure = "The clock ticks steadily. "
  hereWithoutSource = "There is a steady sticking sound. "
  displaySchedule = [2, 2, 4, 4, 8]
  isEmanating = (battery.isIn(clock))
  noLongerHere = "You can no longer here the clicking of the clock. "
;
```

For some purposes, having to define all those different properties for a sound or smell can approach overkill. In such circumstance we can use **SimpleNoise** or **SimpleOdor** instead. For either of these classes we just need to define the **desc** property, and the sourceDesc, descWithSource, descWithoutSource, hereWithSource, and hereWithoutSource properties will all simply display whatever we defined in **desc**. These classes are primarily intended for ambient sounds that are part of a location, for example:

```
cave: Room 'Small Cave'
   "There's hardly enough room to swing a mouse in here, let alone a cat
   (though a mouse o'nine tails would surely be an odd thing to behold).
   Exits lead north, east, southwest and northeast. "
;

+ SimpleOdor 'damp pervasive smell/damp' 'damp smell'
  "There's a pervasive damp smell in the cave. "
;
```

That said, there’s no real reason why we shouldn’t use a SimpleXXXX SensoryEmanation for objects that start in plain view, have ambient sounds, and don’t really need an elaborate range of sound/smell descriptions. For example, if the clock had started on top of the bedside cabinet instead of inside it, we might have defined:

```
+++ SimpleNoise 'steady ticking sound/noise*sounds' 'ticking sound'
   "A steady ticking sound comes from the clock. "
    isEmanating = (battery.isIn(clock))
;
```

The fact that the player would continue to see "A steady ticking sound comes from the clock" if the player character shut the clock in the cabinet would hardly matter, since by then the player would know perfectly well that it was the clock that was ticking.

16.4 Sensory Events

A SensoryEmanation is something ongoing, a smell or sound that persists through time, that can be detected in response to the **smell** or **listen** commands, and that may be reported as part of the ambience of the location. A `SensoryEvent` on the other hand, is an event that occurs at a particular point of time: a sudden flash, a loud bang, or the onset of the smell of burning, for example (although in the last case this is likely to be followed by a smell that persists for some time). A `SensoryEvent` is also an event that other objects in scope (including, but not limited to, actors) can react to.

The library defines three types of `SensoryEvent`: `SightEvent`, `SoundEvent` and `SmellEvent`. When we want a SensoryEvent of one of these three kinds to occur, we simply call its `triggerEvent(source)` method, where *source* is the object in the game we consider to be responsible for the event. The main purpose of the *source* parameter is to determine which other objects (including but not limited to actors) can see, hear, or smell the event in question, but it can also be used by reacting objects to decide how they want to react to the event.

The effect of calling triggerEvent() is simply to notify any observing objects in scope that the event has just occurred (so that they can respond to it). According to the *Library Reference Manual* we should use the mix-in classes `SightObserver`, `SoundObserver` and `SmellObserver` to make an object responsive to SightEvents, SoundEvents, and SmellEvents respectively, but as things stand in the library now, this is not strictly necessary. All these three classes do is to define methods that do nothing; we could just as easily define these methods directly on Actors or other objects we want to react to SensoryEvents without bothering with the mix-in classes. The other point of using these mix-in classes would be (a) in case they become relevant in some future version of TADS 3 and we want to future-proof our code (to the present author this seems unlikely, however) or (b) we particularly want to identify xxxxObservers in our own code, e.g. because we want to write code that loops over every SoundObserver in our game, or we need to test whether something if a SmellObserver for some other purpose than responding to the SmellEvent that's just happened. Again, this seems unlikely to be the case. The upshot is that though in order to be one hundred per cent safe in all eventualities we probably ought to use the `SightObserver`, `SoundObserver`, and `SmellObserver` mix-in classes, in reality we'll be about 99.9% safe if we don't.

What we do need to do, whether or not we use these mix-in classes, is to define the relevant methods on any object we want to respond to SensoryEvents:

- To respond to a `SightEvent`, define notifySightEvent(event, source, info).
- To respond to a `SoundEvent`, define notifySoundEvent(event, source, info).
- To respond to a `SmellEvent`, define notifySmellEvent(event, source, info).

In all three of these methods, *event* is the SensoryEvent object on which triggerEvent(source) is called, *source* is the *source* parameter with which triggerEvent(source) was called, and *info* is a `SenseInfo` object describing the sensory conditions under which the object we're defining this method on can sense *source*. For further details of what a `SenseInfo` object is consult the *Library Reference Manual*, but in the great majority of cases we can probably just ignore the *info* parameter. The kind of case where it might be useful is when a SensoryEvent could occur either right next to the relevant observer or some distance away, and we want the observer to behave differently in the two cases; we could then test whether `info.trans` was, say, `distant` or `transparent`.

The definition of a SensoryEvent can normally afford to be fairly minimal; for example:

```
bangEvent: SoundEvent;
```

We'd then describe the event at the same time as triggering, for example:

```
bomb: Thing 'bomb*bombs' 'bomb'
    explode()
    {
         "There's a loud bang from the bomb! ";
          bangEvent.notifyEvent(self);
    }
;
```

An alternative might be to override the notifyEvent() method to include a description of the event:

```
bangEvent: SoundEvent
   notifyEvent(source)
   {
         "A loud bang comes from <<source.theName>>. ";
         inherited(source);
    }
;
```

This might be preferable if we were going to use `bangEvent` with a range of similar objects, but otherwise the first method is probably the better one to use. Either way we need to ensure that our message describing the event is only displayed if the player character is in a position to sense the event. There are a number of ways we can ensure this, including:

- Check the relative locations of the player character and the source of the SensoryEvent;

- Trigger the SensoryEvent from a SenseFuse or SenseDaemon;
- Use the `callWithSenseContext()` function; e.g. `callWithSenseContext(bomb, sound, {: bomb.explode() } );`

To make an object respond to a SensoryEvent we call its notifyXXXEvent() method, for example:

```
vase: Container 'delicate glass vase*vases'  'vase'
   notifySoundEvent(event, source, info)
   {
      if(event == bangEvent)
        "The vase shakes alarmingly. ";
   }
;
```

A neat way to make an NPC respond to Sensory Events is to have the Sensory Events trigger an InitiateTopic, e.g.:

```
mavis: Person 'aunt mavis/woman*women' 'Aunt Mavis'
   isHer = true
   isProperName = true
   notifySoundEvent(event, source, info)
   {
      initiateTopic(event);
   }
;

+ InitiateTopic @bangEvent
   "<q>Goodness me!</q> cries Aunt Mavis, "<q>Whatever was that!</q> ";
;
```

If a particular SensoryEvent could have more than one source, we could make this scheme a little more sophisticated:

```
mavis: Person 'aunt mavis/woman*women' 'Aunt Mavis'
   isHer = true
   isProperName = true
   eventSource = nil
   notifySoundEvent(event, source, info)
   {
      eventSource = source;
      initiateTopic(event);
   }
;

+ InitiateTopic @bangEvent
   "<q>Goodness me!</q> cries Aunt Mavis, "<q>Whatever was that!</q> ";
   isActive = (mavis.eventSource == bomb)
;

++ AltTopic
  "<q>Be careful what you're doing with that thing!</q> Aunt Mavis cries. "
  isActive = (mavis.eventSource == gun)
;
```

Or we could subclass InitiateTopic:

```
class EventTopic: InitiateTopic
    matchTopic(fromActor, obj)
    {
        local actor = getActor();
        if(eventSource != nil)
        {
           if(eventSource.ofKind(Collection))
           {
             if(eventSource.indexWhich({x: actor.eventSource.ofKind(x)}) == nil)
               return nil;
           }
           else if (!actor.eventSource.ofKind(eventSource))
               return nil;
        }
        return inherited(fromActor, obj);
    }
    eventSource = nil
;
```

Then we could write:

```
+ EventTopic @bangEvent
   "<q>Goodness me!</q> cries Aunt Mavis, "<q>Whatever was that!</q> ";
   eventSource = bomb
;

+ EventTopic @bangEvent
  "<q>Be careful what you're doing with that thing!</q> Aunt Mavis cries. "
   eventSource = gun
;

+ EventTopic @bangEvent
  "<q>Whatever is <<mavis.eventSource.theName>> doing?</q> Aunt Mavis
   complains.
   eventSource = Person
;
```

Another way in which we might typically use a SensoryEvent is to allow an actor to respond to someone knocking at his or her door. Suppose that we have already defined a KnockOn action, then we might define:

```
doorKnockEvent: soundEvent;

modify Door
  dobjFor(KnockOn)
  {
    verify() { logicalRank(120,'door'); }
    action()
    {
     defaultReport(knockNoEffectMsg);
     doorKnockEvent.notifyEvent(otherSide);
    }
  }
  knockNoEffectMsg = '{You/he} knock{s} on {the dobj/him}, but nobody replies. '
;
```

By generating a SoundEvent on the other side of the door, we enable a sound-observing NPC on the other side of the door to respond to the knock if the appropriate notifySoundEvent() method is defined on that NPC.

16.5 Sensory Connections

Sound, light and smells all travel. We have already seen how TADS 3 allows for the travelling of these kinds of sensory data in and out of a closed container by defining various kinds of material, so that, for example, a container that may be opaque to sight may be transparent to sound. But sound, light and smells can travel further than just in and out of containers; under the right circumstances they can travel between rooms. The `SenseConnector` class allows us to define such sense paths between different top-level locations. To do its job, a `SenseConnector` has to exist in all the topic level rooms it connects; it is therefore a subclass of `MultiLoc`. Also, if a `SenseConnector` is actually to do anything, we need to define which senses it can pass and how well it can pass them. There are basically two ways in which we can do this.

First, we can define the SenseConnector's `connectorMaterial` to be one of the existing types of material (`glass`, `paper`, `fineMesh` or `coarseMesh`; there's not much point using adventium since then the SenseConnector won't do anything). If none of those materials do what we want, we can either define a new material that does, or else use the second way.

This second way is to override the SenseConnector's `transSensingThru(sense)` method to return the level of transparency for each *sense*. The return value can be one of `opaque`, `transparent`, `attenuated`, `distant` or `obscured`. So for example to allow a sound-only connection between two adjacent rooms we might define:

```
SenseConnector, Intangible 'wall*walls' 'wall'
   locationList = [bedroom, bathroom]
   transSensingThru(sense)
   {
      return sense == sound ? distant : opaque;
   }
;
```

Note that since SenseConnector inherits directly from the mix-in class MultiLoc, it needs to be mixed-in with some Thing-derived class. Where the SenseConnector is a physical object in the game (such as a window we can see through from both sides), we can mix it in with the object in question. Where it's effectively an abstraction, as here, we can use the Intangible class (this is the main situation in which Intangible is actually useful). We give the SenseConnector the name 'wall' here, not so that the player can refer to is, but so that the parser can (with messages like "You can't reach the gurgling water through the wall"); we simply need to choose a name that makes sense to complete a sentence of the form "You can't reach the x through the..."

There's no requirement that a SenseConnector should connect adjacent locations. We might, for example, want to connect quite remote connections via a video-link:

```
videoLink: SenseConnector, Intang 'video link*links' 'video link'
    locationList = [londonOffice, newYorkOffice]
    connectorMaterial = glass
;
```

To switch this video link on and off we could either switch its connectorMaterial between glass and adventium, or move it in and out of the locations its meant to be linking.

A SenseConnector can also link portable objects; for example two mobile phones which the player character and an NPC carry around with them, allowing them to speak to each other when they're in remote locations. There is, however, a further complication if we want this to be an audio-only link; since the NPC then won't in general be visible to the player character, the NPC won't be in scope. To allow the player to address conversational commands to the NPC over the audio-link we need to put the NPC in scope, probably by overriding **`getExtraScopeItems(actor)`** on the player character's mobile phone to return a list including the NPC while a conversation is in progress via the phone (this would be equally true of any other audio device, of course). This is illustrated in the section on "Remote Sensory Scope" in the article on "Redefining Scope" in the *TADS 3 Technical Manual*.

16.6 The DistanceConnector

There's one type of **`SenseConnector`** that's so commonly useful that it's defined as a class of its own: the **`DistanceConnector`**. This is a **`SenseConnector`** for which **`transSensingThru(sense)`** returns **`distant`** for every sense. It is also already mixed-in with **`Intangible`**. Since it inherits from MultiLoc and we don't need to define any of the common Thing properties on it, a **`DistanceConnector`** may very conveniently be defined using the **`MultiLoc`** template, so a typical **`DistanceConnector`** definition might look like this:

```
DistanceConnector [hallNorth, hallSouth];
```

A DistanceConnector is typically used to connect adjacent locations that are fully visible to each other, such as the two halves of a large room (implemented as two separate Room objects), or the four corners of a town square, or adjacent fields. They can also be used when one location overlooks another, for example a flat roof overlooking some of the area surrounding a building.

A DistanceConnector is really very simple to set up; normally we just have to list the rooms we want it to connect in its locationList property, as we have already seen. The complications start when it comes to how we want things described in the remote loaction; we'll look at that in section 16.8 below. In the meantime there's four more things to note about DistantConnectors.

First, a given room can be included in the locationLists of as many DistanceConnectors as we like, but DistanceConnectors (or rather, the **distant** sensing type which they implement by default) are not transitive. Suppose we were to define:

```
DistanceConnector [field, farmyard];
DistanceConnector [field, riverbank];
```

Then the field and the farmyard would be visible (and audible and smellable) from each other, as would the field and the riverbank, but not the farmyard and the riverbank (normally, this is just what we'd want, since we don't want to end up with everywhere being visible from everywhere else).

Second, subject to certain exceptions to be detailed below, the objects in a remote location joined to the current location via a DistanceConnector can be seen, heard and smelled, but they can't be touched (or tasted), and so they can't be interacted with in any way that would require touch.

Third, by default, the player character can converse with an NPC who is in a remote location joined to the player character's location via a DistanceConnector. If the distance being modelled is more than a short one, this may not be what we want. To change this we can override **canBeTalkedTo(talker, sense, info)** on the NPC in question (or on the Actor class in general). The most useful override might be:

```
modify Actor
    canBeTalkedTo(talker, sense, info)
    {
       return (info.trans == transparent);
    }
;
```

This would allow conversation with an NPC in the same room, or via a SenseConnector that gave a transparent sound connection. For a DistanceConnector that modelled a short distance over which conversation would be possible, we might then define:

```
DistanceConnector [loungeE, loungeW]
    transSensingThru(sense) { return sense == sound ? transparent : distant; }
;
```

And, of course, we could always subclass DistanceConnector (e.g. as ShortDistanceConnector) to do this if we wanted several DistanceConnectors like this.

Fourth, by default the distance modelled by a DistanceConnector is regarded as too far for an object thrown at something in a remote location to reach its target, so it will be reported as falling short (and it will land in the location it was thrown from). If we want to allow an object thrown from one location to hit a target in a remote location, we need to override **checkMoveThrough(obj, dest)** on the DistanceConnector. This might be a good candidate for our custom ShortDistanceConnector class:

```
class ShortDistanceConnector: DistanceConnector
    transSensingThru(sense) { return sense == sound ? transparent : distant; }
    checkMoveThrough(obj, dest) { return checkStatusSuccess; }
;
```

A more sophisticated override might first test the weight and/or bulk of the object thrown (obj) to see if can travel that far, but that's a complication too far for now!

16.7 The Occluder

Once we establish line of sight between two locations, most things in one location should be visible in the other – most things, but not necessarily all. If I'm looking through a window into a room, I may well not be able to see the things inside the room that are up against the wall the window looks through. If a flat roof overlooks the surrounding area, it may be possible to see everything in the surrounding area from the roof, but that doesn't mean that everything on the roof is visible from the surrounding area below.

For this kind of situation we can use an `Occluder`, which can selectively remove from view (or hearing or smell) objects that a SenseConnector has made visible, audible or smellable (or indeed, objects in the same location as the observer, although if we get that effect as well it may well be due to an error in our code). `Occluder` is a mix-in class that can be mixed in with any object in the location whose contents we want to occlude, but it's generally convenient to mix the `Occluder` in with the SenseConnector that establishes the sensory connection in the first place; that way we define both the connection and the exceptions in the same place.

There are two ways we can make an `Occluder` exclude items from view (or hearing, or smell):

1. We can override the Occluder's `occludeObj(obj, sense, pov)` method to return true for every object *obj* we want to be occluded for *sense* from the point of view of the *pov* object (normally an actor doing the sensing; typically the player character). We'd normally use the *pov* object to test whether or not it was in the same location as the object we were thinking of occluding.
2. We can override the `isOccludedBy(occluder, sense, pov)` method on the objects we want to be occluded by *occluder* for *sense* from the point of view of *pov*.

We can also mix these two methods, but in order to do so we must ensure that our overridden `occludeObj()` method on the Occluder ends by returning `inherited(obj. sense, pov)`.

For example, if we want to use a window as a SenseConnector so that we can see into the front room of a cottage from the outside, we might define it thus:

```
Occluder, SenseConnector, Openable, Fixture '(cottage) window*windows' 'window'
  connectorMaterial = (isOpen ? fineMesh : glass)
  locationList = [frontRoom, frontDrive]
  occludeObj(obj, sense, pov)
  {
     if(obj.isOrIsIn(bookcase) && pov.isIn(frontDrive))
       return true;

     return inherited(obj, sense, pov);
  }
;
```

Here we are supposing that the bookcase is up the wall that the window looks through, so that someone peering through the window from the outside (in the frontDrive location) will be unable to see either the bookcase or its contents. We're also supposing that when the window is closed we can only see through it, but that when it's open we can also hear and smell through it.

It's very easy to write buggy `occludeObj()` methods, and the bugs can have quite weird effects (especially if we accidentally occlude the player character or the player character's location). It's hard to give useful general advice on how to avoid bugs here, other than to urge more than usual care and to suggest using `isOccludedBy()` in preference to `occludeObj()` wherever possible (in the above example using `occludeObj()` for the bookcase was nevertheless the right thing to do, since it provides the simplest way of occluding everything in or on the bookcase along with the bookcase itself).

16.8 Describing Things in Remote Locations

16.8.1 Obscured, Remote and Distant Descriptions

As we have seen, setting up a DistanceConnector (or some other kind of SenseConnector) is relatively straightforward, but once we have done so, things in one location become visible from one or more other locations, and the chances are that not only will we want object listings in room descriptions and the like to distinguish objects that are close to hand from those that are in remote locations (i.e. not in the same room as the player character), but we will also want them described differently (depending in part, of course, on how far off we imagine them to be).

TADS 3 provides several mechanisms for controlling how things are listed and described in remote locations (and under other non-standing viewing conditions), so we had better work through them one at a time. Perhaps the best place to start is with the various xxxxSize properties defined on Thing, namely `sightSize`, `soundSize` `smellSize` and `touchSize`. Each of these can have one of three values: `small`, `medium` or `large`; by default most Things are medium in all four senses, though certain classes that are very likely to be used for large objects (Enterable, for example) have a large sightSize by default. Note that none of these properties correlates to the `bulk` of an object; giving an object a `bulk` of 1,000 won't make its `sightSize` large, since these two properties are defined quite independently of each other. The effect of these three values on these three properties is how well objects can be sensed under distant or obscured conditions:

- `small` – the object can't be sensed at all under distant or obscure sensing conditions (for example, it's too small to be seen from a distance).
- `medium` – it's possible to sense that the object is there, and what it is, but not to make out any detail, so the object will be listed but an attempt to examine it will be met with a message like "The whatever is too far away to make out any detail". The actual messages that are used are defined in `defaultDistantDesc` and `defaultObscuredDesc(obs)`, where *obs* is the object (container or SenseConnector) that's doing the obscuring.
- `large` – it's possible both to sense that the object is there and to make out the detail under obscure or distant sensing conditions (e.g, we can make out some details of a house or mountain even though we're some way away from it).

Things can get a bit more complicated than the above schema suggests – if we decide to complicate them (or at least, change them). When a large object is examined under distant or obscured conditions, the description displayed is that defined in its `desc` property (or its `initDesc` property, if applicable). We can, however, change this by defining other properties that take precedence *if* they're defined on an object (by default they aren't); if they are defined, the difference between a medium and large `sightSize` becomes irrelevant (although an object with a small `sightSize` will remain invisible under distant or obscured conditions).

These other properties/methods, in order of the priority they take are:

1. `remoteDesc(pov)` – this is the description that's displayed when this object is viewed from a remote location (i.e. a location other than the one it's in); *pov* is the actor (or other object, e.g. a surveillance camera) doing the looking.
2. `obscuredDesc(obs)` – this is the description that's displayed when the object is viewed under obscured conditions, with *obs* being the object (either a container or a SenseConnector) that's doing the obscuring.
3. `distantDesc –` this is the description that's displayed when this object is examined under distant viewing conditions.

This means that if an object is examined from a remote location, its `remoteDesc()` will be used to provide the description, if a `remoteDesc()` has been defined for it. Failing that, its `obscuredDesc()` method will be used if it has been defined and the relevant SenseConnector results in obscured viewing conditions. Failing that, its `distantDesc` will be used if it has been defined and the relevant SenseConnector results in distant viewing conditions. Failing that its `desc` or `initDesc` property will be used if the relevant SenseConnector results in transparent or attenuated viewing conditions and the `sightSize` is `large`. Failing that either the `defaultDistantDesc` or the

defaultObscuredDesc() will be used. This may look complicated, but it does at least give us a large amount of control (we don't *have* to use most of this if we don't want to, but it's available to us if we do).

There's one further point to underline before we move on, since it's a clear source of potential confusion, and that's the distinction between *distant* and *remote* viewing conditions. Viewing is remote if the actor doing the viewing is in a different top-level room from the object being viewed, that is (in the case that the player character is doing the examining and *obj* is the object being examined) when **gPlayerChar.getOutermostRoom != obj.getOutermostRoom**. Viewing is *distant* if the SenseConnector returns the value **distant** for sight in its **transSensingThrough()** method. This need not be the case; we could set up a SenseConnector between two different top-level rooms that instead returned **transparent**, **attenuated** or **obscured**; viewing conditions would then not be *distant*, though they would still be *remote*.

This distinction is less relevant with sounds and smells, since there is no remoteSoundDesc or remoteSmellDesc. There are, however, **distantSoundDesc**, **distantSmellDesc**, **obscuredSoundDesc(obs)** and **obscuredSmellDesc(obs)** properties and methods, which work in much the same way as their visual equivalents. If an object is listened to or smelled under transparent conditions, or its **soundSize** or **smellSize** (as appropriate) is large, then its associated Noise or Smell object is used if it has one, or otherwise its **soundDesc** or **smellDesc** is used. Otherwise **obscuredSoundDesc(obs)** or **obscuredSmellDesc(obs)** (on the source object, not the SensoryEmanation) is used for objects obscured to sound or smell by *obs*, or **distantSoundDesc** or **distantSmellDesc** is used for objects listed to or smelled at a distance.

16.8.2 Distant, Obscured and Remote Object Listings

We can give an object a paragraph of its own in the listing of objects in room descriptions by giving it a **specialDesc** and/or **initSpecialDesc**, as we have already seen. The second of these, **initSpecialDesc** is used for as long as **isInInitState** is true which, by default, is until the object is moved (although we're free to change this if we want to use some other condition, such as until the item is described). Provided a **specialDesc** is defined we can also define **obscuredSpecialDesc** and/or **distantSpecialDesc** and/or **remoteSpecialDesc(actor)**, where *actor* is the actor doing the viewing (usually the player character). Provided an **initSpecialDesc** is defined we can also correspondingly define **obscuredInitSpecialDesc** and/or **distantInitSpecialDesc** and/or **remoteInitSpecialDesc(actor)**. In both sets the remote version will take precedence when the actor and the object being listed are in different top-level rooms.

By default the **distantSpecialDesc** and **obscuredSpecialDesc** properties copy the

specialDesc property, and the **remoteSpecialDesc()** method displays the **distantSpecialDesc** property (so that if we override **distantSpecialDesc** the change will also be reflected under remote viewing conditions). Likewise the **distantInitSpecialDesc** and **obscuredInitSpecialDesc** by default take their values from **initSpecialDesc**, and **remoteInitSpecialDesc()** displays the **distantSpecialDesc**. We can, of course, override any of this as desired. With a bit more determination and ingenuity we can also override the circumstances under which these properties are used, but that's beyond the scope of this manual; to find up more look up Thing in the *Library Reference Manual* and look at the definitions of methods like **useSpecialDesc()**, **useSpecialDescInRoom(room)**, **useSpecialDescInContents(cont)**, and **useInitSpecialDesc()**.

Where neighbouring locations are connected by DistanceConnectors (or some other kind of SenseConnector), we need to use **distantSpecialDesc** or **remoteSpecialDesc()** alongside **specialDesc** to make it clear when we're listing something that's not in the player character's immediate location, for example:

```
+ table: Surface, Heavy 'large wooden table*tables' 'large wooden table'
    specialDesc = "A large wooden table occupies much of the floor-space at
      this end of the room. "
    remoteSpecialDesc(actor)
    {
       switch(actor.getOutermostRoom)
       {
          case hallSouth:
            "A large wooden table stands at the far end of the hall. ";
            break;
          case carPark:
            "Through the window you can see a large wooden table in the hall. ";
            break;
       }
    }
;
```

Note that in the case of Actors, **specialDesc**, **distantSpecialDesc** and **remoteSpecialDesc(actor)** are farmed out to the current ActorState.

In addition to items that have a **specialDesc** or **initSpecialDesc** defined, which get their own paragraphs in room descriptions, are the miscellaneous portable items which are listed with a single sentence like "You see a glove, a rubber ball, a pair of green Wellington boots, and an old walking stick here". Once again, if some of these items are in a remote location, this needs to be made clear to the player. The library does try to take care of this, but the default results can be rather crude. For example, suppose we define a long hall as two rooms joined by a DistanceConnector, and we give the two ends of the hall the room name 'Hall (east end)' and 'Hall (west end)'. From the west end of the hall we might see a listing like:

```
In the hall (east end), you see a rubber duck.
```

This could be more elegantly phrased, and with other room names the effect can be even more jarring, e.g.:

In the in front of a cottage, you see a brass key.

There are basically two ways we can fix this, either on the remote room we're looking at, or at the room we're looking from. The first of these is generally the simpler, and we'll look at it first.

The way a remote room describes itself under remote viewing circumstances is defined in its **inRoomName(pov)** property, which by default returns **actorInName** (a single-quoted string value). Once again, *pov* is the actor (normally the player character) who's doing the looking. In turn the default definition of **actorInName** is **(actorInPrep + ' ' + theNameObj),** which is what generates a phrase like 'in the hall (east end)' or 'in the in front of a cottage'. The phrase generated can thus be changed by one of three methods

1. Override the **name** and/or **actorInPrep** properties of the room.
2. Override the **actorInName** property of the room.
3. Override the **inRoomName(pov)** method of the room.

Method 1 might be the method to use if we have other reasons for overriding **name** and/or **actorInPrep**. Method 2 is generally the most straightforward method to use, unless we want to vary the locational phrase according to where the remote room is viewed from, in which case we need to use method 3.

For example, in the case of the two-room hall mentioned above, we'd probably use **actorInName**:

```
hallEast: Room 'Hall (East End)' 'the east end of the hall'
    "This large hall continues to the west. A flight of stairs leads up to the
    north. "

    actorInName = 'at the far end of the hall'
    north = hallStairs
    up asExit(north)
    west = hallWest
;
```

On the other hand, if we had a long road with north, mid, and south sections, then we might well use **inRoomName()** on the middle section:

```
roadMid: OutdoorRoom 'Long Road (middle)' 'the middle section of the road'
    "The long road continues to north and south. "
    north = roadNorth
    south = roadSouth

    inRoomName(pov)
    {
       return 'further down the road to the ' + pov.isIn(roadNorth) ? 'south'
        : 'north';
    }
;
```

Whichever way we choose to change the way the remote location is described, the list of miscellaneous portable items in the remote location will always take the form, "*Location phrase (e.g. at the the far end of the hall)* you see *list of items*." If we want to change it further (that is, if we want the list introduced with something other than "you see"), we need to override `remoteRoomContentsLister(other)` on the room we're looking *from.* The simplest way to use this is to make it return a new `CustomRoomLister` to which we can pass the way we want a list of objects in the remote room to be introduced and concluded as the two arguments to its constructor; for example:

```
hallWest: Room 'Hall (West End)' 'the west end of the hall'
    "This large hall continues to the east. A flight of stairs leads down to the
    south. "
    east = hallEast
    south = hallStairsDown
    down asExit(south)
    remoteRoomContentsLister(other)
    {
        return new CustomRoomLister('Right at the far end of the hall {you/he}
            catch{es} sight of', '. ');
    }
;
```

Note that if we do this, whatever we define here will take precedence over whatever we did with `actorInName` or `inRoomName(pov)` on the remote room, the reason being that the `remoteRoomContentsLister()` method is always used when listing miscellaneous portable items in a remote location, while its default behaviour is to return a lister that uses the `inRoomName(pov)` of the remote location whose items are being viewed. By overriding `remoteRoomContentLister(other)` (where *other* is the remote room whose contents are to be listed), we in effect by-pass the use of `inRoomName()`.

Even with this device, we have to introduce the list of items in the remote location with something more or less equivalent to "you see". If we want to introduce it with something equivalent to "there is" or "there are", a `CustomRoomLister` won't quite do the job for us, since it doesn't provide the means to switch between "is" and "are" according to the number of things that are to be listed. But if we really want to do this we can define our own custom Lister class and pass anonymous functions to its constructor, like this:

```
class MyCustomRoomLister: Lister
    construct(prefix, suffix)
    {
        prefixFunc = prefix;
        suffixFunc = suffix;
    }

    showListPrefixWide(itemCount, pov, parent)
        { local p = prefixFunc; p(itemCount); }
    showListSuffixWide(itemCount, pov, parent)
        { local s = suffixFunc; s(itemCount); }
    showListPrefixTall(itemCount, pov, parent)
        { local p = prefixFunc; p(itemCount); }

    /* our prefix and suffix functions */
    prefixFunc = nil
    suffixFunc = nil
;

hallWest: Room 'Hall (West End)' 'the west end of the hall'
    "This large hall continues to the east. A flight of stairs leads down to the
    south. "
    east = hallEast
    remoteRoomContentsLister(other)
    {
        return new MyCustomRoomLister( {itemCount: "At the far end of the
            hall <<itemCount > 1 ? 'are' : 'is'>> " }, {itemCount : ". "} );
    }
;
```

Using a constructor to pass anonymous function pointers to a dynamically created object may seem quite an obscure trick, but it’s potentially quite a powerful one!

Exercise 21: Try implementing a game along the following lines. A town has to be evacuated due to imminent flooding from a nearby river. In one corner of the town square an old woman is sleeping on a bench. The player character, a policeman, has to wake her and persuade her to leave, but she proves resistant to being woken up. The square is large enough to need implementing as four rooms (one for each corner), although they are all visible from one another. An ornamental fountain, into which someone has thrown a coin, stands in the centre of the square, and the water gushing from the fountain makes a sound that should be audible throughout the square. In another corner of the square stands a barrel organ that can be played or pushed around; the player can try playing the organ next to the old woman but this merely makes her wake up for a moment, complain about the noise, and then go back to sleep, as does shouting or blowing a whistle.

Along the north side of the square runs a building that can be entered from the northeast corner of the square. Inside the building are two rooms, a hall and a chamber; the hall is entered from the square, and the chamber has a window (too small to crawl through) overlooking the northwest corner of the square. The chamber contains a radio that can be heard in the hall (when it’s switched on) and in the square (when the window is open). It also contains a whistle. The radio starts out inside a packing case which is opaque to sight but transparent to sound.

From the northwest corner of the square you can go west into a park, which consists of two locations visible from each other. A river (the one that’s about to flood) runs alongside the west side of the park. In the south end of the park (nearest the square) a bonfire is smouldering by the river, letting off acrid-smelling smoke. In the north end of the park (furthest from the square) are a basket of rotting fish (stinking appropriately) and a tall tree with a trumpet caught out of reach in its branches. The branch holding the trumpet can be reached via the ladder that needs to be fetched from the hall. Playing the trumpet in the immediate vicinity of the old woman wakes her up fully and wins the game.

17 Attachables

17.1 The Attachable Framework

In TADS 3 an Attachable is something that can be temporarily attached to something else and later detached from it again. Attachables are potentially one of the trickiest kinds of thing to deal with in TADS 3, not least because after one thing is attached to another, there are so many ways in which the resulting attachment might behave. For example, suppose the player character attaches a rope to a particular object in the current location, and then tries to go to another location while still holding the rope. A number of different outcomes could result, including:

- The object tied to the rope is dragged along behind the player character, so that it ends up in the new location along with the rope.
- The rope is long enough to allow the player character to walk into the new location without dragging the object at the other end of the rope, so that the player character ends up in the new location, with one end of the rope in the player character's hand and the other attached to the rope in the old location.
- The rope is quite short but the object is too heavy to be dragged, so the player character is jerked to a halt as s/he tries to leave the location.
- The rope is quite short but the object is too heavy to be dragged, so that the rope is snatched from the player character's grasp as s/he leaves the location.
- The rope is quite short but the object is too heavy to be dragged, so that the rope breaks as the player character leaves grasping one end of it and enters the new location.

Doubtless one could think of other possibilities, just as one could think of other types of situation in which one object is attached to another. The upshot is that the TADS 3 library can scarcely define a straightforward Attachable class that works absolutely fine straight out of the box; it can only provide a framework which we then have to adapt to the particular circumstance we're trying to model.

There is, however, one adaptation that we can carry out straight away whatever kind of attachment situation we have in mind. Alongside the AttachTo, Detach and DetachFrom actions, for which the library defines framework handling on the Attachable class, the library defines Fasten, FastenTo, Unfasten, and UnfastenFrom actions which the Attachable class ignores. Whatever distinction the library makes between Attach and Fasten, or between Detach and Unfasten, most players will probably regard them as two pairs of synonyms, at least in the case of most of the Attachable objects we're likely to be defining, so we can save ourselves quite a bit of trouble later on if we make them synonyms from the outset by defining:

```
modify Attachable
```

```
   dobjFor(Unfasten) asDobjFor(Detach)
   dobjFor(Fasten)
   {
     verify() { }
     action() { askForIobj(AttachTo); )
   }

  dobjFor(FastenTo) remapTo(AttachTo, self, IndirectObject)
  dobjFor(UnfastenFrom) remapTo(DetachFrom, self, IndirectObject)

;
```

Note that we don't also need to handle `iobjFor()` for these last two commands, since the library assumes that one Attachable can only be attached to another Attachable, so that where a player issues a command of the form **fasten x to y** or **unfasten x from y** then provided x and y are both Attachables we can leave the remapping to the direct object.

This leads into one of the main underlying principles of the Attachable framework; it's designed so that **attach x to y** always means the same as **attach y to x** and we should be able to define the responses to attachment and detachment on an Attachable without worrying about whether it will be used as the direct object or the indirect object.

Note that `Attachable` is a mix-in class that needs to be used in conjunction with Thing-derived classes. The principal Attachable properties and methods we need to know about are:

- `attachedObjects –` a list of objects currently attached to this Attachable; we don't normally need to manipulate this property (since there are methods which do that for us), unless we want certain objects to start out attached at the beginning of the game, but we might want to examine it.
- `beforeTraveler(traveler, connector)` – note that this method is overridden on Attachable to carry out various notifications, so that if we override this method ourselves we'll probably need to call the inherited method somewhere in our code.
- `canAttachTo(obj)` – this should return true if this Attachable can be attached to *obj* (which should also be an Attachable); note that this need only be defined on one of the objects in the potential attachment arrangement. In some complex cases it may be necessary to end by calling the inherited method, but we then need to be careful not to create a deadly embrace in which the direct and indirect objects of at **attach** command keep calling this method on each other.
- `canDetachFrom(obj)` – this should return true if this Attachable can be detached from *obj*. The comments on canAttachTo() apply here also.

- **`cannotDetachMsgFor(obj)`** – the message used to explain why this Attachable cannot be detached from *obj* (which may be nil if the player typed **detach something** without specifying an indirect object). This should return a single-quoted string or a message property pointer.
- **`explainCannotAttachTo(obj)`** – this should display a message (perhaps using reportFailure()) explaining why this Attachable cannot be attached to *obj*.
- **`handleAttach(other)`** – by default this does nothing; we should override it to carry out any further side effects of or display any further messages relating to attaching this object to *other*. Note that this is called on both the direct and indirect objects of an **attach** command, so we don't need to worry about which will turn out to be which (and we should only write our custom handling on one of them). Neither this method nor the next need concern itself with manipulated the attachedObjects list since this is dealt with elsewhere.
- **`handleDetach(other)`** – this is similar to handleAttach(other), except that it is called on both the direct and indirect objects of a **detach** command.
- **`isListedAsAttachedTo(obj)`** – returns true if this item is to be listed as being attached to *obj* when this item is examined; by default this returns true unless this item is permanently attached to *obj* or unless it's the major attachment (see below) in this attachment relationship.
- **`isListedAsMajorFor(obj)`** – returns true if *obj* is listed as attached to this item when *obj* is examined; by default this returns true if this item is the major item in the attachment relationship and *obj* should be listed as attached to this item.
- **`isMajorItemFor(obj)`** – by default attachments are described symmetrically: "attached to the handle is a blade" or "attached to the blade is a handle", but in some cases this would look strange: we'd say "attached to the battleship is a limpet-mine" but hardly "attached to the limpet-mine is a battleship". By default this method returns nil, but where this item is clearly the major item in an attachment relationship (e.g. the battleship when *obj* would be the limpet-mine) it should return true. This only effects the way the attachment relationship is described.
- **`moveWhileAttached(movedObj, newCont)`** – when an attached object is moved to a new container, this method is called on the object and on every object attached to it. *movedObj* is the object that's actually being moved and *newCont* is the container it's being moved into. This can be used, for example, to make all attachments move together (when one is moved into a new container, the rest follow) or else to make the moved object become detached from the object it's attached to. Note, however, that calling **`moveInto()`** from this method on something we're attached to will cause this method to be triggered again, so we must be very careful not to write code here that would get trapped in an infinite

loop; one solution might be to call **baseMoveInto()** on attached objects from this method, thereby avoiding recursive notification calls.

- **travelWhileAttached(movedObj, traveler, connector) –** this is called when this item or anything attached to it is being moved to a new location because it's being carried by *traveler* through *connector*. *movedObj* is the attached object that's actually being carried. By default this does nothing.

For further details on these (and other) method of the Attachable class, look up Attachable in the *Library Reference Manual*.

This all looks complicated enough, but it won't work particularly well out of the box. You can verify this by, for example, defining a pair of lego bricks (one red, one yellow) as Attachables and then try playing with them. You'll find, for example, that you can attach the red brick to the yellow brick, and then carry the red brick into another room, leaving the yellow brick behind while the red brick and yellow brick are still attached to each other! This is because, in the general case, the library simply can't know what rules we want to enforce in such a case: should the red brick have become detached from the yellow brick, or should the yellow brick have come with it?

In this case we'd probably want the bricks to move together when attached, so we could try defining a Brick class that enforces that:

```
class Brick: Attachable, Thing
    canAttachTo(obj) { return obj.ofKind(Brick); }
    moveWhileAttached(movedObj, newCont)
    {
        if(movedObj != self)
            baseMoveInto(newCont);
    }
;
```

If we then define both the yellow brick and the red brick to be of the Brick class, things will work more sensibly (provided we only have two bricks; once we have more than two the above code may not cope). But it might be equally sensible to enforce the condition that a moved brick becomes detached when moved:

```
class Brick: Attachable, Thing
    canAttachTo(obj) { return obj.ofKind(Brick); }
    moveWhileAttached(movedObj, newCont)
    {
        if(movedObj == self)
           foreach(local other in attachedObjects)
            tryImplicitAction(DetachFrom, self, other);
    }
;
```

Although, if that's what we want, we'd probably be better off using a **NearbyAttachable** (see below).

If we have three bricks (adding a green one, say), then our first piece of code won't work if we attach the red brick to the yellow brick, then the yellow brick to the green brick, then take the red brick. The green brick will be left behind. If we try to fix it by changing `baseMoveInto()` to `moveInto()` to trigger the notifications on the green brick, we'll cause a stack overflow. One possible solution would be to define a custom property (let's call is `notificationCount`) on the `Brick` class which is increased by one every time we enter the `moveWhileAttached()` routine and decremented by one every time we leave it. We can then use `moveInto()` within `moveWhileAttached()` to trigger a train of notifications, but check the value of the `notificationCount` property to check that the object `moveWhileAttached()` is being called on hasn't already handled the notification:

```
class Brick: Attachable, Thing
    canAttachTo(obj) { return obj.ofKind(Brick); }
    moveWhileAttached(movedObj, newCont)
    {
        notificationCount++;
        if(movedObj != self && notificationCount == 1)
            moveInto(newCont);

        notificationCount--;

    }
    notificationCount = 0
;
```

We could go on adding examples, but it would be impracticable to try to cover every possible case, and too long-winded to try to cover even every supposedly common case. The three examples given above indicate what kinds of thing we may need to think about in dealing with Attachables; we should next move on to some of the special kinds of Attachable that the library already defines for us.

17.2 NearbyAttachable

A common requirement with Attachables is that if we attach one object to another, the two objects should be in the place (that is, within the same container). The `NearbyAttachable` class enforces this condition. More specifically it enforces:

- When one `NearbyAttachable` is being attached to another, both must be in the same container; if necessary one or the other object is moved to that container by an implicit action (by default the indirect object's container is the one used).
- When one `NearbyAttachable` that is attached to another is removed from their common container, the two become detached.

Simply using the `NearbyAttachable` class rather than `Attachable` can thus solve a lot of problems for us. Provided this is actually the sort of attachment relation we want, we can use `NearbyAttachable` straight out of the box. For example, we could just define our `Brick` class as:

```
class Brick: NearbyAttachable, Thing
    canAttachTo(obj) { return obj.ofKind(Brick); }
```

```
;
```

If **NearbyAttachable** only does more or less what we want, but not quite, there are ways in which can customize how it behaves. In particular, we can override the **getNearbyAttachmentLocs(other)** method, which defines where the attached objects should end up when the object the method is defined on is attached to *other*. This method returns a list with three values: **[loc1, loc2, priority]**.

- **loc1** is the location where we want this object to end up after the attachment.
- **loc2** is the location where we want the *other* object to end up.
- **priority** is an integer value; if one **NearbyAttachable** is being attached to another then **getNearbyAttachmentLocs()** is called on both; the one which returns the higher **priority** is the one that determines what the target destinations of the two objects will be. If the two priorities are equal, the indirect object wins. If one object is a **NearbyAttachable** and the other isn't (we can attach a **NearbyAttachable** to another kind of **Attachable**) then the **NearbyAttachable** wins (since the other object won't be offering any suggestions in any case).

By default, **getNearbyAttachmentsLocs(other)** returns **[location, location, 0]** unless *other* is non-portable, in which case it returns **[other.location, other.location, 0]** (since *other* then can't be moved). If we want to override this method, we probably want the third value in the returned list to be something greater than 0 to ensure that our override wins out over the default.

For example, suppose we want to tweak our Brick class so that if the player character is holding one brick and attaches it another (or attaches the other brick to the brick he's holding), both bricks end up being held by the player character. We could do it like this:

```
class Brick: NearbyAttachable, Thing
    canAttachTo(obj) { return obj.ofKind(Brick); }
    getNearbyAttachmentLocs(other)
    {
        local priority = isIn(gPlayerChar) ? 10 : 0;
        return [location, location, priority];
    }
;
```

Alternatively, we might want to so arrange it so that both bricks end up being held by the player character wherever they start out:

```
class Brick: NearbyAttachable, Thing
    canAttachTo(obj) { return obj.ofKind(Brick); }
    getNearbyAttachmentLocs(other)
    {
        return [gActor, gActor, 10];
    }
;
```

Clearly, this only scratches the surface of what we *could* do with this method. In principle we could make attached objects end up anywhere we liked (provided it's accessible), including two different locations (despite the fact that the class is called `NearbyAttachable`). The main limitation is that taking (or otherwise moving) a `NearbyAttachable` while it's attached to something else results in its becoming detached, which may not always be what we want. However, if its behaviour for attaching things is what we want (and as we've just seen, it can be pretty flexible), then it may we be worth using a NearbyAttachable and overriding its `movedWhileAttached()` method (perhaps along the lines of the last example in the previous section).

17.3 Other Kinds of Attachable

17.3.1 PlugAttachable

`PlugAttachable` isn't exactly a kind of Attachable, it's a mix-in class we can use with other kinds of Attachable to make the command **plug into** and **unplug** or **unplug from** behave like **attach to** and **detach** or **detach from**. For example, to make a plug that can be plugged into an electrical socket, we could just do this:

```
+ socket: Attachable, Fixture 'socket*sockets' 'socket'
    isMajorItemFor(obj) { return obj == plug; }
;

+ plug: PlugAttachable, NearbyAttachable, Thing 'plug*plugs' 'plug'
    canAttachTo(obj) { return obj == socket; }
;
```

17.3.2 PermanentAttachment

A `PermanentAttachment` is an object that's described as attached to something else but can never become detached. The object it's attached to should also be a `PermanentAttachment`. One of the object should list the other in its `attachedObjects` list (the game will then make the relationship symmetrical at startup).

Alternatively we can use `PermanentAttachmentChild`. If one object effectively contains the other (because the other is a effectively a component of the first), we can define the containing object a `PermanentAttachment` and the contained object as a `PermanentAttachmentChild`, and the two will automatically link up their attachment lists at start up. This could be done with something like:

```
+ broom: PermanentAttachment, Thing 'wooden broom*brooms' 'wooden broom'
    "A brush is attached to one end of the broom. "
    contentsListedInExamine = nil
;
```

```
++ brush: PermanentAttachmentChild, Thing 'brush*brushes' 'brush';
```

Whether this achieves a great deal that couldn't be achieved by making the brush a Component and perhaps customizing a few of its message properties is not all that clear. At any rate note that `PermanentAttachment` and `PermanentAttachmentChild` are mix-in classes that must be mixed in with a Thing-derived class, and that defining `contentsListedInExamine = nil` on the parent object, or `isListedInContents = nil` on the child object, is generally a good idea if we don't want to see the child object described as being in the parent object (e.g. "a broom (in which is a brush)").

The main functional difference between using these classes and Component are the different responses they give to **attach** and **detach** commands (e.g. "The brush is already attached to the broom" as opposed to "You cannot attach that to anything").

17.3.3 SimpleAttachable

`SimpleAttachable` is not a class defined in the library, but a subclass of Attachable defined in the SimpleAttachable.t extension that should be in the ../lib/extensions directory of your TADS 3 installation, and which also comes with the attachment.t sample game that forms the basis of the next exercise.

The `SimpleAttachable` class is meant to make handling one common case easier, in particular the case where a smaller object is attached to a larger object and then moves round with it.

More formally, a `SimpleAttachable` enforces the following rules:

1. In any attachment relationship between SimpleAttachables, one object must be the major attachment, and all the others will be that object's minor attachments (if there's a fridge with a red magnet and a blue magnet attached, the fridge is the major attachment and the magnets are its minor attachments).
2. A major attachment can have many minor attachments attached to it at once, but a minor attachment can only be attached to one major attachment at a time (this is a consequence of (3) below).
3. When a minor attachment is attached to a major attachment, the minor attachment is moved into the major attachment. This automatically enforces (4) below.
4. When a major attachment is moved (e.g. by being taken or pushed around), its minor attachments automatically move with it.
5. When a minor attachment is taken, it is automatically detached from its major attachment (if I take a magnet, I leave the fridge behind).
6. When a minor attachment is detached from a major attachment it is moved into the major attachment's location.

7. The same **SimpleAttachable** can be simultaneously a minor item for one object and a major item for one or more other objects (we could attach a metal paper clip to the magnet while the magnet is attached to the fridge; if we take the magnet the paper clip comes with it while the fridge is left behind).
8. If a **SimpleAttachable** is attached to a major attachment while it's already attached to another major attachment, it will first be detached from its existing major attachment before being attached to the new one (ATTACH MAGNET TO OVEN will trigger an implicit DETACH MAGNET FROM FRIDGE if the magnet was attached to the fridge).
9. Normally, both the major and the minor attachments should be of class **SimpleAttachable**.

Setting up a **SimpleAttachable** is then straightforward, since all the complications are handled on the class. In the simplest case all the game author needs to do is to define the **minorAttachmentItems** property on the major **SimpleAttachable** to hold a list of items that can be attached to it, e.g.:

```
    minorAttachmentItems = [redMagnet, blueMagnet]
```

If a more complex way of deciding what can be attached to a major **SimpleAttachable** is required, override its i**sMajorItemFor()** method instead, so that it returns true for any *obj* that can be attached, e.g.:

```
    isMajorItemFor(obj) { return obj.ofKind(Magnet); }
```

One further point to note: if you want a Container-type object to act as a major **SimpleAttachment**, you'll need to make it a **ComplexContainer**.

Exercise 22: The player character is the only survivor aboard a small scouting space-ship that has just been attacked, holing its hull so that all the air is evacuated, killing everyone else aboard. The player character has survived since he was suited up making repairs to the antenna on the outside of the ship when the attack occurred. The attacking vessel has departed, but now the player character must effect repairs to take his own ship to safety.

The player character starts the game in the airlock. He is wearing a space suit to which is attached an air cylinder (nearly exhausted) and a helmet; a lamp is currently plugged into the helmet but can be unplugged from it and plugged in elsewhere for recharging. Just as the game begins, the lights aboard ship go out, indicating a power failure. The outer and inner airlock doors are operated by levers, with dials indicating the air pressure outside the hull, inside the airlock, and inside the ship.

Just inboard of the airlock is a storage chamber with a rack containing a spare air cylinder (full enough to last the whole game and more), a charging socket, an equipment locker, a freezer, and a winch (fixed in place). Inside the equipment locker are two connectors (for joining lengths of cable), a short length of cable, and a roll of hull repair fabric. To operate the winch while the main power supply is out, it is necessary to attach one end of the cable to the winch and the other to the charging socket. The charging socket can also be used to recharge the lamp, when the lamp is plugged into it. A hawser runs from the winch; one end of the hawser can be carried by the player character into other locations (in which case there'll be a length of hawser running all the way back to the winch); if the free end of the hawser is attached to something (such as a pile of debris) and the winch then operated (by pressing a button on it while the winch has power), the hawser will be rewound, dragging whatever's attached to the other end with it.

Aft of the storage hold is the Engine Room. Here there's a power switch that can be turned on to restore power to the whole ship, but only once the main fault has been restored, and an airflow control lever than can be pulled to repressurize the ship, but only once the hole in the hull has been repaired.

Forward of the Storage Hold is the Living Quarters, which took the brunt of the blast from the attacking vessel, and now has a large hole in part of the hull. This can be repaired by attaching a square of fabric from the locker to the hull. To one side of the Living Quarters is the door into a cabin, but this won't open until pressure has been restored to the ship. Along the floor of the Living Quarters is an electrical conduit which contains the main power cable for the ship (now exposed by the same blast that tore through the hull). A length of the cable has been burned away, and must be repaired by attaching the short length of cable from the locker to the two ends of the severed cable by means of the electrical connectors (also from the locker). Unfortunately, the mass of debris left over from the blast blocks access to the conduit to repair the cable, and can only be removed by using the winch and hawser.

Once the hull has been patched the ship can be repressurized (using the lever in the Engine Room), and once the ship has been repressurized the cabin door can be opened, allowing access to the cabin. Inside the cabin is a bed and a cabinet, the latter containing a security card.

Forward of the Living Quarters is the Bridge, containing (amongst anything else you think should be on the Bridge of a scouting space-ship) a card reader and green button. If the green button is pressed once main power has been restored and the security card is attached to the card reader, the controls come back to life, and the game is won.

This is probably the trickiest exercise in this manual. Once you've spent as much time as you want to on it, take a look at the attachment.t sample game.

18 Menus, Hints and Scoring

18.1 Menus

There are various places in a work of Interactive Fiction where it can be useful to display a menu, usually at the beginning (perhaps in response to an **about** command if we have a lot of information to offer our players) and perhaps at the end, if, for example, we want to offer a number of options in response to an **amusing** command.

We can construct a menu in TADS 3 by using a combination of `MenuItem` and `MenuLongTopicItem` objects. We create the structure of the menu with a tree of `MenuItem` objects. We can use `MenuLongTopicItems` at the ends of the branches to display substantial amounts of text.

On a `MenuItem` we normally only need to define the `title` property (as a single-quoted string); this is the title of the option as it appears in its parent menu. It's also the heading given to the menu when the `MenuItem` displays its own list of options, unless we override the `heading` property to do something different. So for example we could define:

```
+ MenuItem
      title = 'Instructions'
      heading = 'Instructions Menu'
;
```

Or, using the MenuItem template, simply:

```
+ MenuItem  'Instructions' 'Instructions Menu';
```

Within a MenuItem (using the + notation) we can either place more MenuItems (to implement sub-menus) or MenuLongTopicItems, which will actually display some text (or do whatever else we want them to do). We define the `title` and `heading` properties of a MenuLongTopicItem in the same way as for a MenuItem. We also define the `menuContents` property to display some text or do whatever else we want to do. This can be a string (single-quoted or double-quoted) to be displayed, or a routine that does whatever we want. If it's a single-quoted string, this can be the last item in the template. If we want a sequence of MenuLongTopicItems to function as a series of 'chapters', then we can override their `isChapterMenu` property to be true (this causes a 'next' option to be displayed at the end of each MenuLongTopicItem which the player can select to proceed directly to the next MenuLongTopicItem without having to go back to the parent menu).

In order to get a menu displayed in the first place, we call the `display()` method on the top level menu we want to display. If we do so in response to a command, it's a good idea to display a brief message like "Done." immediately afterwards.

So, for example, to display an "About" menu (in response to an **about** command), we could do something along the following lines:

```
versionInfo: GameID
   ...
   showAbout()
   {
     aboutMenu.showAbout();
     "Done. ";
   }
;

aboutMenu: MenuItem 'About';

+ MenuLongTopicItem 'About this game'
  'This is the most exciting game I have ever written (not that that\'s
   saying much). The protagonist is an entrant into the South Dakota
   Annual Paint Drying Contest. Consequently the special command <b>watch
   paint dry</b> is one you\'ll need to make frequent use of in this game. '
;

+ MenuLongTopicItem 'Credits'
   menuContents() { versionInfo.showCredit(); }
;

+ MenuItem 'How to Play' 'Playing Instructions';

++ MenuItem 'Instructions for Players New to IF'

+++ MenuLongTopicItem 'Standard Commands'
   'LOOK, INVENTORY, blah, blah, QUIT'
;

+++ MenuLongTopicItem 'Movement Commands'
   'NORTH, SOUTH blah blah...'
;

+++ MenuLongTopicItem 'Conversational Commands'
   'ASK FRED ABOUT PAINT, TELL BOB ABOUT PAINT, blah blah...'
;

++ MenuLongTopicItem 'Instructions for Player New to <i>Paint Dry!</q>'
   'Use the command <b>watch paint dry</b>. Then use it again. And again.
    And again and again and again and again.'
;
```

There's just one point to note: some TADS 3 interpreters can't cope with more than nine items under a single menu, so it's as well to design our menus so that they don't display more than nine items at a time. If we need more options than that, then we should put them under sub-menus.

If we want to include one top-level menu among the options of another menu, we can do so by explicitly listing it in the `contents` property of the menu we want it to display under. For example, suppose `intructionsMenu` is the top-level menu displayed in response to an **instructions** command, but we also want to be able to offer this instructions menu from within the about menu. We can do this by defining:

```
aboutMenu: MenuItem 'About'
  contents = [instructionsMenu]
;
```

Whatever options/sub-menus we list explicitly in the `contents` property will be listed in addition to whatever options we define under the menu with the + notation.

We can control the order in which menu items are displayed by overriding their `menuOrder` property. Items are sorted in ascending order of this property just before the menu is displayed; by default `menuOrder` is set to `sourceTextOrder` (the order in which the menu items are defined within the same source file).

If we want to add an item to a menu during the course of play, we can do so by calling the `addToContents(obj)` method on the `MenuItem` to which we want to add *obj*. To remove *obj* from a menu during the course of play use `contents -= obj`.

Finally, there is an instructions menu built into the library that's just a little tricky to access. The library file instruct.t defines an **instructions** command, which by default does a huge text dump. It can, however, be made to present the same set of instructions in the form of a menu (`topInstructionsMenu`). To do this we need to do a complete recompile for debugging after ensuring that the constant INTRUCTIONS_MENU is defined. In Workbench, go to Build -> Settings from the menu and click Defines in the dialogue box; then add INSTRUCTIONS_MENU to the list of symbols to define; then use the Build -> Full Compile For Debugging option from the main Workbench menu. If compiling from the command line using t3make, add -D INSTRUCTIONS_MENU to the command line (or project file) and use the -a option the first time you recompile.

18.2 Hints

There is, of course, no need to provide any hints at all in a work of Interactive Fiction; whether or not to do so depends on the nature of our game, our target audience, and our own sense of what makes our game complete. If we do decide to provide hints, there's obviously a number of ways in which we can do it. The way provided by the library is an invisiclues type system (in which a series of progressively clearer hints on any given topic can be revealed one at a time) embedded in a context-sensitive hint system (in which the topics on which hints are offered become available and cease to be available depending on their relevance, in relation to where the player is in the game). If we don't want this hint system at all, we can exclude the file hintsys.t from our build. In what follows, however, we shall assume we do want to use the hint system built in to the TADS 3 library.

This hint system is basically a specialization of the menu system discussed in the previous section. We construct a hint system for our game by creating a menu of hints, or rather a menu of goals that the player may want information on at any particular time. Our top-level hint menu should be an object of the `TopHintMenu` class (and there should only be one of these defined in our game). An object of this class automatically registers itself as the root menu of the hint system, and will thus be the menu that's invoked when the player issues a **hint** command. We only need to give this menu object an object name if we want to refer to elsewhere in our code, for example to make it accessible as an option from the **about** command (by listing it in the `contents` property of another menu).

Located in the `TopHintMenu` we should put either `HintMenu` objects (if we want to create submenus in our hint system), `HintLongTopicItem` objects (the hint system equivalent of `MenuLongTopicItem`, which we might use, say, for a permanent set of instructions on using the hint system) or `Goal` objects (which we'll say more about below). The *only* difference between `HintMenu` and `TopHintMenu` is that the latter automatically registers itself as the root of the hint menu tree.

The difference between a `HintMenu` and a `MenuItem` is that the former is intended to part of an *adaptive* menu system. A `HintMenu` is only displayed when it has active contents, and its `contents` property only holds its active contents. An item is active if its `isActiveInMenu` property is true. This property is true by default for `HintLongTopicItem`, and for a `HintMenu` with active contents, and for an open `Goal`. For the most part game authors don't need to worry about this as the library takes care of it all. We can just define our menu tree in much the same way as we would for ordinary menus, except that most of the items at the bottom of the tree will be Goals:

```
+ TopHintMenu 'Hints';

++ HintMenu 'In the Garden';

+++ Goal 'How do I water the exotic cabbage?';
+++ Goal 'How do I reach the tower window?';

++ HintMenu 'In the Bedroom';

+++ Goal 'How do I tell which woman is the sleeping princess?'
+++ Goal 'Where can I hide from the jealous prince?'
```

Whether we need HintMenus between the TopHintMenu and the Goals depends on how many Goals are likely to be active at any one time. If we're confident it will never be more than nine, then we probably don't need any intermediate sub-menus. If it may be more than nine, then we need to implement some kind of sub-menu structure, since some TADS 3 interpreters can't cope with menus that have more than nine items.

Setting up the menu structure is relatively straightforward; the real work of building an adaptive hint system in TADS 3 comes with defining the various Goal objects. These are objects that represent the various objectives the player may be trying to pursue at particular points in the game. A Goal thus consists primarily of the question to which the player is looking for an answer (e.g. 'How do I get past the five-headed cat?'), defined in its `title` property, and a list of hints relating to the question, defined in the menuContents property. Since these two properties are common to all Goals, we can define them via the Goal template, e.g. :

```
+ Goal 'How do I get past the five-headed cat?'
    [
       'Why is the cat such a problem? ',
       'What else might keep its mouths occupied? ',
       'What do cats like to chase? ',
       mouseHint,
       'You\'ll need five mice, of course, one for each mouth. '
    ]
;
```

These hints will be displayed one at a time, as the player requests each in turn. For the most part, they can just be single-quoted strings, in which case they'll just be displayed. But if we want the displaying a hint to have some further side-effect, we need to use a Hint object (which is what we are assuming mouseHint to be in the above example).

The default behaviour of a Hint object is to display the text in its hintText property (a single-quoted string) and to open the Goals listed in its referencedGoals property (opening a Goal makes it available to the player, as we shall see below). In this example the mouseHint Hint may suggest to the player that s/he needs to find some mice, which will then make finding mice a new Goal for the player to pursue. There was no point displaying this new Goal before, since that would be a potential spoiler for the cat puzzle, but once we offer a hint that the cat may be distracted by mice, it's fair enough to offer another series of hints about finding mice. Since the hintText and referencedGoals properties are so commonly defined on Hint objects, they can be defined via a template:

```
++ Hint 'Perhaps the cat would be distracted by mice. ' [mouseGoal] ;
```

Here mouseGoal would be another Goal object that we'd define elsewhere. Putting two plus signs before the Hint object implies that we're locating it inside the previous Goal; there is no need to do this, but doing so does no harm, and it's a convenient way of keeping the Hint close to its associated Goal in the source code without it interfering with the hint containment hierarchy.

If we want displaying a Hint to carry out any other side-effects besides opening one or more Goals, the best place to code them is in the Hint's getItemText() method. If we override this method we must remember to conclude it with return inherited; or else make it return a singe-quoted string containing the text of the hint.

As we've already mentioned, but not yet fully explained, Goal objects are used to create an *adaptive* hint system, that is a system that displays hints only when they become relevant, and removes them once they cease to be relevant. To that end, a Goal can be in one of three states: UndiscoveredGoal, OpenGoal or ClosedGoal, the current state of a Goal being defined by its goalState property. A Goal generally starts out in the UndiscoveredGoal state (although we could define it as being an OpenGoal if we want it to be available at the start of play). A goal is undiscovered when it concerns an objective the player doesn't yet know about (so to display it would be at best an irrelevance and at worst a spoiler). Once the player has reached a point in the game when a particular Goal becomes relevant, it changes to the OpenGoal state. Open goals are those that are displayed in response to a **hint** command, since they represent the problems the player is currently working on (or could be working on) at that point in the game. Once the player achieves the objective defined by the Goal, the Goal is no longer relevant, so it changes to the ClosedGoal state. Every time the game is about to display a hint menu it runs through the Goals contained in that menu to see which should be changed to OpenGoal and which should be changed to ClosedGoal. It then displays all those for which goalState is OpenGoal.

It is, of course, up to us to define under what conditions our Goals become opened and closed. We can do this by means of the following properties:

- **openWhenAchieved** – the goal becomes open when this Achievement object (see the next section, on scoring) is achieved (we set this property to the Achievement object in question).
- **openWhenDescribed** – the goal becomes open when this object has been described (i.e. when the player has examined it).
- **openWhenKnown** – the goal becomes open when this Topic or Thing becomes known to the player (i.e. gPlayerChar.knowsAbout(openWhenKnown) becomes true).
- **openWhenRevealed** – a single-quoted string value; the goal becomes open when this tag is revealed (e.g. if this were set to 'cat' then the goal would become open when gRevealed('cat') became true).
- **openWhenSeen** – the goal becomes open when this object has been seen by the player character.
- **openWhenTrue** – the goal becomes open when this condition becomes true; this can be used to define any condition that doesn't fit the other openWhenXXXX properties. For example, if this goal should become open when the player has seen the blue plaque and taken the brass key, then we could define openWhenTrue = (bluePlaque.seen && brassKey.moved).

Note that the goal will be opened when *any* of the above are satisfied, so, for example, if we defined:

```
+ Goal 'How do I get past the five-headed cat?'
    [
       'Why is the cat such a problem? ',
       'What else might keep its mouths occupied? ',
       'What do cats like to chase? ',
       mouseHint,
       'You\'ll need five mice, of course, one for each mouth. '
    ]
    openWhenSeen = cat
    openWhenKnown = mice
    openWhenRevealed = 'five-cat'
    openWhenDescribed = bewareOfTheCatSign
;
```

Then this Goal would become open *either* when the player character had seen the cat, *or* when the player character knows about the mice *or* when the 'five-cat' tag has been revealed *or* when the player character has examined the bewareOfTheCatSign.

Goals are closed by a similar set of properties: **closeWhenAchieved**, **closeWhenDescribed**, **closeWhenKnown**, **closeWhenRevealed**, **closeWhenSeen**, and **closeWhenTrue**, which all work in the same way as their openWhenXXXX equivalents. Thus a more typical Goal definition might look like:

```
+ Goal 'How do I get past the five-headed cat?'
    [
       'Why is the cat such a problem? ',
       'What else might keep its mouths occupied? ',
       'What do cats like to chase? ',
       mouseHint,
       'You\'ll need five mice, of course, one for each mouth. '
    ]
    openWhenSeen = cat
    closeWhenTrue = (cat.curState == chasingMiceState)
;
```

Behind this series of **openWhenXXX** and **closeWhenXXX** properties are a pair of properties called simply **openWhen** and **closeWhen**. If we wanted to, we could use these to extend the set of conditions a Goal can test for. For example, suppose in our game we quite often wanted to open and close goals when certain items were moved, then we could modify Goal to allow this:

```
modify Goal
   openWhenMoved = nil
   closeWhenMoved = nil
   openWhen = (inherited || (openWhenMoved != nil && openWhenMoved.moved))
   closeWhen = (inherited || (closeWhenMoved != nil && closeWhenMoved.moved))
;
```

Then we'd be able to use our new **openWhenMoved** and **closeWhenMoved** properties along with all the others.

18.3 Scoring

Many works of Interactive Fiction, especially more story-based ones, don't need to keep score. If scoring is irrelevant to our game, we can simply exclude the file score.t from our game, and all traces of score-keeping will be removed. If, however, we do want to keep a score in our game, then there are several ways we can go about it.

If all we want to do is to keep a record of the points the player has scored, we can simply add them to `libScore.totalScore` (the total number of points scored so far). So, for example, to award two points, we could simply write:

```
libScore.totalScore += 2;
```

More commonly, though, we want to tell the player not only how many points have been scored, but what they have been awarded for. One way we can do that is by calling the function `addToScore(points, desc)` where *points* is the number of points we want to award and *desc* is either a description of the achievement (as a single-quoted string) or an `Achievement` object. For example we might write:

```
addToScore(2, 'Distracting the cat');
```

Note that if we call `addToScore(points, desc)` more than once with the same *desc*, it will be considered the same achievement (i.e. it will appear only once when the player's achievements are listed in response to a **full score** command), although the points associated with it will be increased accordingly.

We can also use `Achievement` objects to award points, and this probably gives us the greatest degree of control over how scoring works in our game. One big advantage of using `Achievement` objects is that they can track how often they've been used to award points, which makes it quite straightforward to avoid awarding the player repeatedly for the same action. Another big advantage is that, under certain circumstances, we can use `Achievement` objects to calculate the maximum score in our game automatically.

To award points using an `Achievement` object, we can use one of the following methods:

- `addToScoreOnce(points)` – award *points* points for this Achievement, provided we haven't previously awarded any points for it (in which case the score remains unchanged). If points are awarded, return true, otherwise return nil.
- `awardPoints()` - award the number of points defined in this Achievement's `points` property.
- `awardPointsOnce()` - award the number of points defined in this Achievement's `points` property provided no points have been awarded for this Achievement before; return true if points were awarded.

In addition, we can define or query the following properties for an Achievement:

- **desc** – a double-quoted string or a routine that displays a string describing this Achievement.
- **maxPoints** – the maximum number of points that can be awarded for this Achievement. By default this is the same as points, but we may need to use a larger value here if we are going to allow this Achievement to be awarded more than once.
- **scoreCount** – the number of times points have been awarded for this Achievement
- **totalPoints** – the total number of points that have been awarded for this Achievement

In the simplest and most common case, we expect to award points for each Achievement only once, in which case the only properties that need concern us are **desc** and (possibly) **points**. We can also look at **scoreCount** to see if the Achievement has been achieved. Since **desc** and **points** are the commonest properties to award on an Achievement, they can be defined via a template. The points property is optional in the template, but if it is present it should come before the **desc**, and the number of points immediately preceded by a + sign. So, for example, we could define:

```
catAchievement: Achievement  +2 "distracting the cat";

mouseAchievement: Achievement  "catching some mice";
```

Callining **catAchievement.awardPointsOnce()** would then award two points for distracting the cat. For **mouseAchievement**, though, we should need to call **mouseAchievement.addToScoreOnce(2)** or whatever, since no points property has been defined. If there are five mice to catch and the player gets one point for each mouse, we might go for a more elaborate definition of **mouseAchievement**:

```
mouseAchievement: Achievement
    "catching <<mouseCount()>>. "
    mouseCount()
    {
        if(scoreCount > 1)
           "<<spellInt(scoreCount)>> mice";
        else
           "a mouse"
    }
    maxPoints = 5
    points = 1
;
```

If we ensure that *all* the points in our game are awarded through calling **awardPoints()** or **awardPointsOnce()** on Achievement objects, *and* that we adjust **maxPoints** appropriately on any Achievement for which points can be awarded more than once, then we can leave the library to calculate the maximum number of points in our game (provided, that is, that the winning path through the game causes all the Achievement objects to be awarded). If there are alternative paths through the game which would result in the awarding of points through different Achievements, then we would need to use the **addToScoreOnce()** method (or the **addToScore()** function) to award points on those alternative Achievements, and we would also need *not* to define their points properties, in order to ensure that they did not get added to the total points available. Ensuring that all winning routes through our game ended up with the same maximum score could prove quite tricky, and may or may not be possible depending on the details of the game design; having an automatically calculated maximum score may work best with a highly linear game with one set of Achievements that must be met.

If we're not confident that the library can calculate the maximum score for us, or we're not sticking to restrictions that allow it to do so, we need to calculate it for ourselves (or find out what it is by playing through our game and seeing what it comes to), and then override **gameMain.maxScore** with whatever the maximum score is.

There's one more property on **gameMain** we may want to override in relation to scoring, and that's **scoreRankTable**. This is the property to use if we want a message like 'This makes you a total novice' appended to the player's score. The property should consist of a list of entries, each of which is itself a two element list, the first element being the minimum score required to attain the rank, and the second being a string describing that rank, for example:

```
gameMain: GameMainDef
   ...
   scoreRankTable = [
     [ 0, 'a total novice'],
     [ 10, 'a well-meaning tyro'],
     [ 25, 'a casual adventurer'],
     [ 50, 'a would-be hero'],
     [ 100, 'a paladin']
   ]
;
```

As in the foregoing example, the table must be arranged in ascending order of scores. If we want to change the wording of the message that announces the rank from the standard "This makes you..." form, we can do so by overriding **libMessages.showScoreRankMessage(msg)**, e.g.:

```
modify libMessages
    showScoreRankMessages(msg) { "This gives you the rank of <<msg>>. "; }
;
```

Exercise 23: This final exercise will give you an opportunity to brush up on EventLists and one or two other things from earlier in the manual, as well as menus, hints, and scoring.

The map for this game is fairly simple. Play starts in 'Deep in the Forest' from which paths lead northeast, southeast and west. To the west is a dead end (blocked by a fallen tree), but attempting to travel down it the first time results in the player character finding a branch, which he takes. In the starting location itself a variety of forest sounds can be heard, or small animals seen moving about. Northeast from the starting location is the 'By the River' location. From here paths run southwest, back to the starting location, and southeast. Progress north is blocked by the stream. If the player attempts to cross the stream, the first two attempts are blocked with suitable messages, but the third time the attempt is allowed, and the player character drowns. This happens whether the player types **north** or **swim river**, but the first two refusal responses should differ according to the command used. Thick undergrowth prevents walking along the bank of the stream to east and west. There should be a suitable selection of riverside atmospheric messages.

Southeast of 'By the River' is 'Outside a Cave', from which paths run northwest (back to the river) and southwest (to a clearing). The cave lies to the east. The cave is in darkness, and the only way out from it is to the west. The second time the player character leaves the cave there's a warning message about an imminent rockfall. The third time the player leaves the cave the rockfall occurs, blocking the entrance to the cave. Inside the cave is a bucket, but the player character can't find it until light is brought into the cave.

Southwest of 'Outside a Cave' is a clearing, from which paths run northeast (back to 'Outside a Cave') and northwest (back to the starting location, 'Deep in the Forest'). In the clearing a large bonfire is billowing clouds of acrid smoke, the smell of which is described as increasingly overwhelming the longer the player character remains in the clearing. The player character needs to leave the clearing to the south in order to find the way back to the car park and win the game, but the smoke keeps driving him back. There's also a very simple NPC who starts out in this location, a tall man who walks round and round the four locations in the forest, but stops for one turn each time he reaches the river to scoop up some water in his hands.

To win the game the player needs to light the branch from the bonfire to make it act as a torch, then go to the cave to collect the bucket, fill the bucket with water from the river, then pour the water on the bonfire to douse it, and finally leave the clearing to the south.

Provide the game with a help/about menu, a set of adaptive hints, and a score for each step. Make the game automatically calculate the maximum score.

When you've got as far as you want to with your attempt, compare it with the event (eventful walk) sample game.

19 Beyond the Basics

19.1 Introduction

We've now covered all the basics of writing Interactive Fiction in TADS 3; if you've mastered everything up to now you should be well on your way to being able to carry out most common TADS 3 programming task without too much difficulty. But the nature of IF programming means we often want to carry out less common programming tasks; it's likely to be the unusual that makes our game stands out.

We can't cover everything TADS 3 can do here, but we can briefly survey some of the other features that are likely to be of interest. In the present chapter we shan't try to explain them in any detail; we'll simply introduce them and give some brief pointers (in particular to where more information can be found).

19.2 Parsing and Object Resolution

19.2.1 Tokenizing and Preparsing

Parsing is the business of reading the player's command, matching it to an action the game can execute and deciding which game objects it refers to; in case of ambiguity object resolution is the business of narrowing down the possibilities to those that are most likely. The standard cases have already been dealt with in the two chapters on Actions.

The first stage of interpreting a player's command is to divide it up into tokens (roughly speaking, the individual words that make up the command, so that the command **take the knife** contains the tokens 'take', 'the' and 'knife'). To do this the parser uses the `Tokenizer` class. For details of how this works and how it can be customized, see the chapter "Basic Tokenizer" in Part VI of the *System Manual*. If *str* is a string we want to tokenize, we can do so with a statement like:

```
local toks = Tokenizer.tokenize(str);
```

If we then wanted to execute this as it were a command issued by the player, we could do so with:

```
executeCommand(gPlayerChar, gPlayerChar, toks, true);
```

For some information on what the parser goes with the tokens when matching player input to action syntax and noun phrases, you could take a look at the article on GrammarProd in Part IV of the *System Manual*, but it's not the easiest read, and it's usually possible to be able to do what you want in TADS 3 without understanding that part of the system in any depth. The *Technical Manual* article on "The Command

Execution Cycle" gives a rough outline of the parsing stage of command execution, but again it's hardly required reading at this point.

It is sometimes useful to be able to intercept the player's input and tweak it before the parser gets to work on it. For this purpose we can use a `StringPreParser`. This class performs its work in its `doParsing(str, which)` method, where *str* is the string (initially the command typed by the player) containing the command we may want to tweak. This method should return either the original or an adjusted string, or nil. If it returns a new string, this will be used instead of the command that was entered. If it returns nil, then the command will be aborted (on the assumption that the `StringPreParser` has fully dealt with it). For example, if we wanted to write a particularly prudish game, we could define a StringPreParser like this

```
StringPreParser
  doParsing(str, which)
  {
     if(str.toLower.find('shit'))
     {
         "If you're going to use language like that I shall ignore you! ";
          return nil;
     }
     return str;
  }
;
```

We can have as many StringPreParsers as we like in our game, and the player's input will be processed by each in turn (unless one of them returns nil, in which case processing of the player's command will stop there). We can control the order in which StringPreParsers are used by means of their `runOrder` property. For more details, look up StringPreParser in the *Library Reference Manual*.

One type of tweaking that could often be useful to apply to a player's command before parsing it would be to correct any typos. If this is something you're interested in adding to your game, you might want to check out Steve Breslin's spellingCorrector extension, which can be downloaded from the IF-Archive.

19.2.2 Object Resolution

The standard TADS 3 library can understand a reasonable range of noun phrases as referring to a particular object: zero or more adjectives followed by a noun, or a pair of nouns separated by 'of', or a noun followed by a number, or a locational phrase like "the ball on the table", but we may have objects in our game that don't fit any of those patterns, such as "Cranky the Clown" or "S and P magazine" or "the path to the west". For ways of dealing with such non-standard noun phrases, see the article on "Handling Odd Noun Phrases" in the *TADS 3 Technical Manual*.

We've already seen that we can bracket selected words in the `vocabWords` property of an object to make them weak tokens, which can't then match the object on their own (but only in conjunction with other tokens that are not weak). There are a few more special characters we can include in a **vocabWords** property.

The hash symbol **#** can be used to match any number in the player's input (whether entered as a numeral or as a word). So, for example, if we defined the following:

```
+ locker: OpenableContainer: Fixture '# locker*lockers' 'locker'
;
```

This would match 'locker 1' or '2 locker' or 'locker nine' or 'locker 999' and so on. Note that *any* number defined as an adjective in the **vocabWords** property of an object can come either before or after the noun in the player's input (since noun phrases like 'locker 2' or 'box three' or 'room 5' are fairly common, and '5 room' wouldn't look right).

We can also use "\u0001" to match any literal adjective in the player's input (that is, an adjective the player explicitly types in quotation marks), so, for example, the following:

```
+ door: Door '"\u0001"' door'  'door'
;
```

Would be matched by the commands **x "red" door** or **x 'bumbleweed' door** but not **x red door**.

An asterisk * can be used as a wildcard that will match anything at all, but it has to be explicitly assigned to the object's noun property. For example, this object will respond to absolutely anything the player calls it:

```
+ Thing
    name = 'ubiquitous object'
    noun = '*'
;
```

Though we'd probably want to use **matchNameCommon()** (see below) to narrow down what this matched.

If we want to define adjectives that an object can match but no noun, we can use a dash (**-**) in the noun slot of the **vocabWords** property. This is particularly useful when an object inherits from a class that already defines nouns and just wants to define adjectives (remember that the **vocabWords** defined on an object are in *addition* to any it inherits from its superclasses).

If these various ways of using the **vocabWords** property of a simulation object don't give us enough control over which objects are matched, we can override **matchNameCommon()** on the object in question (roughly speaking, this is the equivalent of an Inform 6 **parse_name** routine). For details of how to do this, look up **matchNameCommon()** (defined on **VocabObject**) in the *Library Reference Manual*. To make sense of it you'll also need to look at **matchName()** in the same place. For example, suppose we have a car key object that we don't want to be matched by either ‘car’ or ‘key’ but only by the phrase ‘car key’; we could define:

```
+ carKey: Key 'car key*keys' 'car key'
    matchNameCommon(origTokens, adjustedTokens)
    {
        /*
         *   We're looking for the exact phrase 'car key' which the player
         *   will only have typed if adjustedTokens is ['car', &adjective,
         *   'key', &noun] with possible variations in the case of 'car' and
         *   'key' and whether 'key' is singular or plural.
         */

        if(adjustedTokens.length == 4
            && adjustedTokens[1].toLower == 'car'
           && adjustedTokens[3].toLower is in ('key', 'keys'))
            return self;

        return nil;
    }
;
```

Once the parser has identified all the simulation objects in scope that could match the noun phrase entered in the player’s command, it constructs a *resolve list* containing these objects. If there’s more than one object in this list, the library calls `filterResolveList()` (also defined on `VocabObject`) on each of the objects in the list. If we wish, we can override this method on any of the objects and return a completely new list of objects. One situation in which this might be useful is where a noun phrase could refer to an object the player may not know about yet, for example, consider the following:

```
+ redDoor 'red door*doors'  'red door'
   "It has a brass knocker on it. "
;

++ brassKnocker: Component 'brass knocker*knockers'  'brass knocker'
;

+ blueDoor 'blue door*doors' 'blue door'
  "It has a silver knocker on it. "
;

++ silverKnocker: Component 'silver knocker*knockers'  'silver knocker'
;
```

If the player examines the red door and issues the command **x knocker** without having examined the blue door, the player would almost certainly mean the brass knocker, so the parser's question ("Which knocker do you mean, the brass knocker or the silver knocker?") could look a little strange (at this point the player doesn’t even know there *is* a silver knocker). One way to deal with this would be to override `filterResolveList()` on the two knocker objects so that each knocker removes itself from the resolve list until its door has been examined (which we can test by looking at the `described` property of the door).

For more details on how this method works, look up `filterResolveList()` in the *Library Reference Manual*.

A slightly different situation is where the player is well aware (or ought to be well aware) of a number of objects that could match what s/he types, but is more likely to mean one thing than another. For example, suppose that the player character is carrying around a red book and a blue book, each of which she's likely to need to consult quite frequently. Since **x book** won't select between the books, the player is quite likely to type **x red** to refer to the red book. On occasion, there may well be other red objects in scope, but in this scenario it's still more likely that **red** by itself is intended to refer to the red book, so it will be most helpful to the player if we can nudge the parser towards preferring the red book to other red objects (other things being equal) while telling the player what choice the parser has made in cases of potential ambiguity. For this purpose we can use the `vocabLikelihood` property; the default value is 0, so we could give the red book a `vocabLikelihood` of 10, say, to make the parser prefer it in cases of ambiguity it couldn't resolve in any other way. For further details, look up `vocabLikehood` (defined on `VocabWords`) in the *Library Reference Manual*.

The techniques outlined above all presuppose that we're happy with the parser's idea of scope (since the parser will only match objects it considers to be in scope for the current command). In slightly simplified terms, scope defines what objects the player character can actually interact with for a given command; normally this will be the objects the player character can see (or perhaps sense in some other way), the main exception being that we can obviously talk, read or think about things without being able to see them. The vast majority of the time, the parser's idea of scope will do what we want, but every now and again we may want something different (e.g. to allow the player character to converse with a distant actor via a telephone). For information on adjusting scope, see the article on "Redefining Scope" in the *TADS 3 Technical Manual*.

19.3 Similarity, Disambiguation and Difference

One of the things no player of our game wants to see is a disambiguation disaster like this:

```
>take mat
Which mat do you mean, the mat, the mat or the mat?
```

We should normally be able to avoid this sort of thing by ensuring that we give each object a unique name property, in this case perhaps 'rubber mat', 'beer mat' and 'place mat', but there may be cases where we don't want to do this, perhaps because the full distinctive name of the object would seem too cumbersome for use in inventory listings and the like (perhaps our game contains a large black decorative door mat, a medium-sized black decorative door mat and a large brown decorative door mat). In such cases we can instead define the **disambigName** property to contain a uniquely distinctive name. Then the regular name will be used in inventory and room listings, while the **disambigName** will be used in disambiguation prompts like the one above. If we do define a **disambigName** we should be careful to ensure that it does actually provide a combination of words that identifies each object uniquely, and that this combination of words will match the **vocabWords** property of the object.

If we want we can define the order in which items are listed in a disambiguation prompt using the **disambigPromptOrder** property, which by default takes its value from the **pluralOrder** property. This can be useful when items are explicitly enumerated, or otherwise have some natural order, e.g.:

>open drawer
Which drawer you mean: the top drawer, the middle drawer or the bottom drawer?

For further details, look up these properties in the *Library Reference Manual*.

Some objects may be genuinely indistinguishable, at least for the purposes of our game. One pound coin or silver dollar is much like another. Each grape in the bunch or apple in the orchard may be effectively identical. When we want items to be effectively identical we should (a) assign them to the same class and (b) set their **isEquivalent** property to true. For example, to define a bowl containing five identical grapes we might write:

```
class Grape: Food 'green round grape*grapes*fruit'  'grape'
    "It's round and green. "
    isEquivalent = true
;

bowl: Container 'bowl*bowls' 'bowl'
;

+ Grape;
+ Grape;
+ Grape;
+ Grape;
+ Grape;
```

Then we'll see the bowl described as containing five grapes (rather than "a grape, a grape, a grape, a grape and a grape"), and players will be able to issue commands like **take a grape** without the parser bothering them with a disambiguation prompt. One thing to watch out for with equivalent objects is irregular plurals, when we'll need to override the **pluralName** property. For example, if we're writing a game set on a farm, and we're defining an Ox class and a Sheep class with **isEquivalent** set to true, we'd better define their **pluralName** properties as 'oxen' and 'sheep' so we don't end up with messages like "In the large field you see six oxes and eight sheeps."

Even when objects are equivalent in this sense, the parser can distinguish them in some cases; for example the player could refer to "a grape in the bowl" or "the grape on the table" even when the grapes are otherwise identical. For more details on how the parser distinguishes objects in this and other cases look up the `distinguishers` property Of Thing and the `Distinguisher` class in the *Library Reference Manual*.

Before the parser distinguishes by location, it tries to distinguish by ownership. This can be defined explicitly by setting the `owner` property (so that, for example, Bob's wallet remains Bob's wallet even if Nancy steals it), or implicitly by location, (so that, for example, a pen in Bob's inventory can be referred to as Bob's pen if it's not explicitly owned by anyone else). For more details look up the `owner`, `getNominalOwner()` and `canOwn(obj)` properties on Thing in the *Library Reference Manual*.

19.4 Fancier Output

We have already seen how we can use HTML tags to format the output of a game; for example "<b>...</b>" to display text in bold, or "<FONT COLOR=RED>...</FONT>" to display text in red. But HTML-TADS can do a great deal more than that; in addition to the other things we can do with HTML mark-up we can display pictures and play sound. For details see the *Introduction to HTML TADS*.

A normal TADS 3 game window has two areas: the status line at the top, and the main text area underneath. While this is both the traditional layout for works of IF and also fine for most purposes, sometimes people want something a little different, for example a special window to display graphics, or maybe windows to display the player character's current inventory or a list of commands. The TADS 3 Banner API lets us divide the main game window up any way we like. It's described in the "Banner Window Display Model" chapter in Part VI of the *System Manual*, and many of the functions used to control the Banner API are explained in the "tads-io Function Set" chapter in Part IV of the *System Manual*. Using these functions to control a multi-windowed output is not quite as simple as those two chapters may make it appear, however; if you want to use the Banner API in your own game you should also read the article on "Using the Banner API" in the *TADS 3 Technical Manual* and consider using the `CustomBannerWindow` class, which is described in that articled and supplied in customBanner.t extension which you should be able to find in the ../lib/extensions file of your TADS 3 installation.

If you want to use any of these fancy output features (HTML mark-up, sound, graphics, and banners) you need to be aware that not all TADS 3 interpreters may be able to handle them (especially the TADS 3 interpreters available on non-Windows systems). Your game therefore needs to test the capabilities of the interpreter it's running on and provide some suitable alternative for features a less capable interpreter cannot support. For example, if your game displays a picture in response to an **examine** command, it should also display a textual description on interpreters that can't cope with graphics. To determine what features the interpreter your game is running on can support you can use the `systemInfo()` function, which is fully described in the "tads-io Function Set" chapter in Part IV of the *System Manual*.

There are some thing we can do to improve the output of a TADS 3 game that don't much depend on the capabilities of particular interpreters. The textual output of a TADS 3 game doesn't get written straight to the screen; it's buffered by something called the *transcript*, which is basically a Vector of objects encapsulating the strings that are due to be displayed. Of course these strings eventually *are* displayed, but not before the library gets a chance to run through the transcript and tidy things up first. This, for example, is the mechanism that gives us implicit action reports like "(first unlocking the door, then opening it)" rather than a series of separate reports. With a little bit of effort and ingenuity we can also intervene in the transcript to improve the output of our game before it's actually displayed. For example, instead of:

>east
Bob stands up.
Bob follows you out.

We could have:

>east
Bob stands up and follows you out.

Or instead of:
>take three coins
gold coin: Taken
gold coin: Taken
gold coin: Taken

We could have:

>take three coins
You take three gold coins.

For an explanation of how to achieve these effects, see the article on "Manipulating the Transcript" in the *TADS 3 Technical Manual* (it's worth pointing out, however, that manipulating the transcript can be quite tricky, so you may want to get some experience doing other things with TADS 3 before you attempt this). If you haven't already done so you should read the article on "Some Common Input/Output Issues" in the *Technical Manual*. In particular you need to be aware that there are occasions when your fancy output (or input) can be defeated by the text-buffering that the transcript is performing. In such cases you need to deactivate the (text-buffering) transcript before doing your fancy stuff and reactivate it afterwards, with a coding pattern than typically looks like:

```
gTranscript.deactivate();
/* do fancy stuff here */
gTranscript.activate();
```

Another kind of output we may occasionally want to tweak is that from implicit action reports. Most of the time what the library does is fine, but every now and then we may feel we want to tweak something like "(first dropping the priceless antique vase)" into "(first putting the priceless antique vase carefully down on the floor)". To find out how, see the article on "Implicit Action Reports" in the *TADS 3 Technical Manual*.

As we've seen the library changes two successive dashes into an en-dash and three into an em-dash. This is great for producing nice-looking textual output, but can be problematic if we actually want to see a run of dashes (for example, because we're trying to display some kind of diagram). The place where the library converts runs of dashes is the **typographicalOutputFilter,** so if we need to disable this behaviour from time to time to draw our diagrams, we can override it thus:

```
modify typographicalOutputFilter
    isActive = true
    activate() { isActive = true; }
    deactivate { isActive = nil; }
    filterText(ostr, val) { return isActive ? inherited(ostr, val) : val; }
;
```

The we can call **typographicalOutputFilter.deactivate()** before outputting our dash-using diagram and **typographicalOutputFilter.activate()** afterwards. Depending on the context we may also need to call **gTranscript.deactivate()** and **gTranscript.activate()** before and after outputting our diagram.

Something simpler we can do to improve the appearance of out textual output is to include the cquotes.t extension in our game build; this will convert all single straight quote marks (') into curly quotes ('), or typographical quotation marks, if you prefer. If we don't do this we'll either have to fill our textual output with lots of HTML entities such as **&rsquo;**, which look a little ugly in source code and are tedious to type, or define our own output tag, or put up with an ugly mixture of straight apostrophes in our own messages and curly ones from the library messages, which will look quite poor.

19.5 Changing Person, Tense, and Player Character

Interactive Fiction is normally narrated in the second person singular and the present tense: "You are carrying a spade, a compass, and an old brown sack" or "You see nothing of interest in the old brown sack." But we can if we like change both the person and the tense. In addition to second-person narration, TADS 3 allows narration in either the first person ("I see nothing of interest in the old brown sack") or the third ("Martha sees nothing of interest in the old brown sack"). It's also possible to write a game in the past tense ("You saw nothing of interest in the old brown sack"), or partly in the present and partly in the past (perhaps using the latter for flashbacks).

To change to first-person or third-person narration is fairly straightforward; we just need to override the **pcReferralPerson** property on the player character object (typically called **me**). The default value is **SecondPerson**; for first-person narration we simply change it to **FirstPerson**, and for third-person narration we change it to **ThirdPerson**.

If we are using third-person narration there's one more step we need to take: we need to give the player character object a name by which he or she will be referred to, for example:

```
me: Actor 'martha/woman*women'  'Martha'
    isProperName = true
    isHer = true
    pcReferralPerson = ThirdPerson
;
```

If we have written all our own action response messages in the form "{You/he} turn{s} the handle" rather than "You turn the handle", then they will automatically adapt to whichever person we use; we could even switch between first, second and third-person narration in the course of the game and all our messages would automatically adapt. This is one reason why it's a good idea to get into the habit of writing all our message using parameter substitution strings; if we do that and then decide half way through the game that our seemingly cool idea for third-person narrative isn't working out so well after all, all we have to do is to change **pcReferralPerson**; if we hadn't used parameter substitutions we'd also have to go back and change all our custom messages by hand .

Changing tense is also reasonably easy. If we want to narrate our entire game in the past tense then all we need to do is to override **gameMain.usePastTense** to true (which will take care of all the library messages) and then write all our own text in the past tense. If we want to switch tenses during the course of our game, things get a little more complicated. For details of how to handle this, see the article on "Writing a Game in the Past Tense" in the *TADS 3 Technical Manual*.

If we want to change player character during the course of a game, we can do this simply with the **setPlayer(actor)** function. For example, if we start out with **me** as the player character, and later want to switch to Mary as the player character, we can just call **setPlayer(mary)**; we could subsequently call **setPlayer(me)** to switch back to the original player character.

Actually, we probably need to do a *bit* more than that, since while **setPlayer()** indeed changes the player character, it doesn't give any indication to the player that it has done so, so at the very least, we might want to display some text informing the player about the switch; it would probably be a good idea to look around from the perspective of the new player character too, e.g.:

```
        setPlayer(mary);
```

```
      "All of a sudden, you find you are Mary!\b";
      mary.lookAround(true);
```

Something else we can do is to add an indication in the status line that the player character has changed; that's often done by adding " (as so-and-so)" after the room name. We can achieve that with the following modification to BasicLocation:

```
modify BasicLocation
    statusName(actor)
    {
        inherited(actor);
        if(actor != me)
            "  (as <<actor.theNameFrom(actor.name)>>)";
    }
;
```

We have to use the rather convoluted `actor.theNameFrom(actor.name)` rather than just `actor.theName` here otherwise we'd just see "(as you)" in the status line, which wouldn't be particularly informative.

One other thing we need to take care of once we start swapping the player character around is the way the actor object in question is described when it's the player character, and when it's an NPC (as it will be if the player character encounters it when the player character is someone else). The Actor class defines the `desc` property to show `pcDesc` when the Actor is the player character and `npcDesc` otherwise. For NPCs that remain NPCs throughout the game it doesn't really much matter whether we define the `desc` property or the `npcDesc` property, since the end result will be the same (though if we use a template to define an Actor we're in fact defining the `npcDesc` property), just as for a player character that remains a player character throughout the game it makes no practical difference whether we define the `desc` property or the `pcDesc` property; but if our game does allow the player character to switch between actors, then the distinction between these properties becomes relevant.

19.6 Making Use of Room Parts

Very early on we mentioned in passing that Rooms are defined with a number of room parts. A Room comes with a defaultFloor, defaultCeiling, defaultEastWall, defaultWestWall, defaultNorthWall and defaultSouthWall. An OutdoorRoom comes with a defaultGround and a defaultSky. The room parts associated with an location are stored in its `roomParts` property (which is thus a list of objects). This set up allows these commonly use room part objects to be employed in multiple locations without our having to create a separate set of objects for each location.

There are various ways in which we might want to customize room parts. Some rooms may not have a complete set of room parts; for example, if we're in the middle section of a passage running from north to south there won't be a north wall or a south wall, so we might define:

```
passageMid: Room 'Long Passage'
    "This long passage continues to north and south. "
    north = passageN
    south = passageS
    roomParts = static inherited - defaultNorthWall - defaultSouthWall
;
```

Examining room parts will generally result in bland descriptions like "You see nothing special about the north wall." One way to change this for a particular location is to define one or more special room parts for that location and use them in place of the defaults, for example:

```
loungeNorthWall: defaultNorthWall
   desc = "The north wall is covered with chintzy wallpaper. "
;

lounge: Room 'Lounge'
  roomParts = static inherited - defaultNorthWall + loungeNorthWall
;
```

If, however, we're then going to define a separate Decoration object to represent the wallpaper, we needn't also create a custom wall object. Instead we can make the Decoration a `RoomPartItem`, associate it with the north wall, and then give it a specialDesc which in effect describes the north wall:

```
+ RoomPartItem, Decoration 'chintzy green wallpaper' 'chintzy wallpaper'
   "It's a particularly vile shade of green. "
   specialNominalRoomPartLocation = defaultNorthWall
   specialDesc = "The north wall is covered with chintzy wallpaper. "
;
```

Examining the north wall will give the specialDescs of any objects associated with it via the `specialNominalRoomPartLocation` property. These will appear in place of the default non-description of the wall, so we effectively get a customized description of the wall without having to use a new wall object.

We don't have to make something a RoomPartItem to associate it with a particular room part. Any thing in a room can have its `specialNominalRoomPartLocation` or its `initNominalRoomPartLocation` set to one of that room's room parts, and the object will then be listed when the appropriate room part is examined. Making something a RoomPartItem means that it will *only* be listed when its associated room part is examined, and won't also be listed in a room description. So, for example, if there's a picture hanging on the east wall, we can make it a RoomPartItem if we don't want it mentioned until the player examines the east wall, but we'd just make it an ordinary Thing if we wanted it also mentioned in a room description; perhaps:

```
lounge: Room 'Lounge'
;
```

```
+ portrait 'picture/portrait*pictures portraits' 'portrait'
  "The portrait depicts a very stern-looking Victorian gentleman. "
  initSpecialDesc = "A portrait hangs on the east wall. "
  initNominalRoomPartLocation = defaultEastWall
;
```

In this case the portrait will be mentioned both in the room description and when the east wall is examined, until the portrait is moved.

Normally room parts are fairly static things, but it's possible that we may occasionally need to remove or add room parts during the course of a game, for example if a wall is demolished, or if a wall acts as a sliding partition that can be opened and closed. If we need to add or remove room parts from a particular location we should call the methods `moveIntoAdd(room)` or `moveOutOf(room)` on the room part in question.

It's occasionally useful to determine which of a location's room parts is the floor for that room. To find that, we can use the `roomFloor` property of the location in question (note this should be treated as a read-only property; to change the floor of a room we need to manipulate the `roomParts` property).

For more details, look up RoomPart and RoomPartItem in the *Library Reference Manual*. You might also want to look up the initNominalRoomPartLocation and specialNominalRoomPartLocation properties on Thing.

19.7 Pathfinding and Timekeeping

There's at least a couple of situations where it can be useful to have a game calculate a path from one location to another on our game map. The first is where we want NPCs to be able to find their way from A to B so they can go off on some errand, or come looking for the player character, or do whatever else they need to do when it involves their being in some other place than where they are now. The other is when we want to implement a **go to x** command for the player, and we need to work out how to get the player character from his/her current location to x. To do either of these (or any related pathfinding jobs) we can use the pathfind.t extension which should be in the ../lib/extensions directory under your main TADS 3 Author's Kit directory.

Perhaps a less common requirement is to have some way of relating the passing of time (as measured by clocks, watches and other timepieces we actually implement as game objects) to key events in our game. For example, if we want to arrange it so that the significant meeting with Bob takes place at noon and the player character's arrival at the lighthouse occurs at 2 pm, and we've equipped our player character with a watch, then we'll probably need some means of having the time shown on the watch advance from noon to 2 pm between these two events without reaching 2 pm until the player character reaches the lighthouse. For this kind of purpose, you might want to check out the subtime.t extension that's also in the ../lib/extensions directory.

19.8 Coding Excursus 18

Although we've covered most of the coding topics needed for most of what you're likely to need to do in TADS 3, and although you can always find out about the rest by reading the *TADS 3 System Manual*, there's a couple more topics we should mention in particular, if only to point out which other parts of the *System Manual* are most worth studying.

19.8.1 Varying Argument Lists

We talked about defining methods and functions very early on, but one thing we didn't mention is that both methods and functions can be defined to take a variable number of arguments. There are two ways to define such a function or method. We can either write:

```
myFunc(someArg, ...)
{
   /* code */
}
```

Or else

```
myFunc(someArg, [args])
{
     /* code */
}
```

Here either the ellipis (...) or the [args] parameter can be replaced with any number of arguments (including none) when this function is called, so that any of the following would be legal ways of calling **myFunc()**:

```
myFunc(2);
myFunc(2, 'foobar');
myFunc(2, me, 3, 'ridiculous-looking pants');
```

And indeed, many other variants besides. If we use the ellipsis notation, then to obtain the values of the arguments within the function or method we must use the **getArg(n)** function, where *n* is the number of the argument we want to manipulate. The number of arguments is then **argcount**. For example, with the third call to **myFunc()** in the above example, **argcount** would be 4, **getArg(1)** would be 2, **getArg(2)** would be **me**, **getArg(3)** would be 3, and **getArg(4)** would be 'ridiculous looking pants'. If we use the second form, with **[args]** in place of the ellipsis (we can use any name here, it doesn't have to be 'args'), then we can obtain the variable arguments directly by looking at the **args** list within the function (or method); for example in the same example **arg[1]** would be me, **arg[2]** would be 3, and **arg[3]** would be 'ridiculous looking pants'.

We can use any number of fixed parameters (like **someArg** in the above example) before the ellipsis or list notation, included no fixed parameters at all.

We can also *send* a list value to a function or method, as though the list were a series of individual argument values. To do this, place an ellipsis after the list argument value in the function or method call's argument list:

```
local lst = [me, 3, 'sensible-looking shirt'];
myFunc(2, lst...);
```

This passes four arguments to **myFunc()**, not two.

For a fuller (and probably clearer) explanation of this, see the sections on "Varying parameter lists" and "Varying-argument calls" in the chapter on "Procedural Code" in Part III of the *TADS 3 System Manual*.

19.8.2 Regular Expressions

It's possible to do quite a bit of string manipulation and matching with the ordinary string methods, such as **find()** and **findReplace()**, and for certain purposes these can be fine. For example, if we need to check whether the player's command includes the name 'Nathaniel Weatherspoon' and replace it with 'nate' we can perfectly well convert it to lower case and use the **find()** method, e.g.

```
StringPreParser(str, which)
  doParsing(str, which)
  {
     if(str.toLower.find('nathaniel weatherspoon'))
        str = str.toLower.findReplace('nathaniel weatherspoon', 'nate',
          ReplacAll);

     return str;
  }
;
```

In more complex cases we may soon run up against the limitations of this method. For example, if we wanted to test whether the player's command contained any of the prepositions 'at', 'in', 'on', or 'by', this would be much trickier, since we should not only need to look for each of them individually, but also check that they were occurring as a word in their own right, and not as part of some other word, such as 'attack' or 'indecent' or 'honest' or 'ruby'; simply surrounding them with spaces wouldn't work either, since them we'd miss any of these words if the occurred right at the end beginning of the end of the string we were testing ('he wanted to pass them by' or 'on the table is a brass bell'). For this kind of case we're really far better off searching with the aid of a regular expression such as:

```
rexSearch('<NoCase>%<(at|in|on|by)%>', str);
```

This will test whether any of 'at', 'in', 'on' or 'by' occur as separate words in **str**, without worrying about whether they're in upper or lower case.

At first sight, regular expressions look horrifyingly confusing creatures. At second and third sight, they're merely confusing (if we're not used to them). But if we plan to do a significant amount of string manipulation they're worth getting to grips with sooner or later. To find out about them, read the "Regular Expressions" chapter in Part IV of the *TADS 3 System Manual*. Then read it again, and expect to have to refer to it frequently until you become *very* familiar with regular expressions. And when you start using regular expressions in your own code, start simple and don't be too disheartened if your first efforts don't quite work as you expect. In the long term it *is* worth getting to grips with these beasties.

19.8.3 LookupTable

There's one other class we'll briefly mention here, the **`LookupTable`**. This can be used like a list or Vector, except that we can index it on any kind of value. We create a new LookupTable in much the same way as we create a new Vector:

```
    myTab = new LookupTable();
```

Once the LookupTable has been created we can store and retrieve values in and from it using arbitrary keys, for example:

```
    myTab['villain'] = bob;
    myTab[bob] = myrtle;
    myTab[[myrtle, 'mood']] = 'sad';
```

We can then retrieve these values with:

```
    local v1 = myTab['villain'];
    local v2 = myTab[bob];
    local v3 = myTab[[myrtle, 'mood'];
```

Following which v1, v2 and v3 would be bob, myrtle and 'sad' respectively. If we try to assign a value to a key that already exists, we'll simply override the key-value pair with a new one. If we try to retrieve a value for a key that hasn't been defined, we'll get the value nil.

For more details, see the chapter on LookupTable in Part IV of the *TADS 3 System Manual.*

19.8.4 Multi-Methods

TADS 3 has a new feature called "multi-methods." This implements a relatively new object-oriented programming technique known as multiple dispatch, in which the types of *multiple* arguments can be used to determine which of several possible functions to call. The traditional TADS method call uses a *single-dispatch* system: when we write **`x.foo(3)`**, we're invoking the method foo as defined on the object x, or as inherited from the nearest superclass of x that defines that method. This is known as single dispatch because a single value (x) controls the selection of which definition of foo will be invoked. Multiple dispatch extends this notion so that multiple values can be considered when selecting which method to invoke. For example, we could write one version of a function `putIn()` that operates on a Thing and a Container, and another version of the same function that operates on a Liquid and a Vessel, and the system will automatically choose the correct version at run-time based on the types of *both* arguments; e.g. (leaving a lot to the imagination):

```
putIn(Thing obj, Container cont)
{
    obj.moveInto(cont);
    "{You/he} put{[s]|} <<obj.theName>> into <<cont.theName>>";
}

putIn(Liguid liq, Vessel ves)
{
    local blk = liq.bulk;
    liq.bulk -= ves.getFreeBulk();
    if(liq.bulk <= 0)
        liq.moveInto(nil);
    vess.addLiquid(liq, blk);
    "{You/he} pour{s/ed} <<liq.theName>> into <<ves.theName>>. ";
}
```

For more details, see the chapter on Multi-Methods in Part III of the *System Manual*.

20 Where To Go From Here

If you've followed *Learning TADS 3* this far, you're well on your way to knowing all you need to know to write your own games in TADS 3. "Knowing all you need to know" is not, however, the same thing as committing the entire contents of this manual to memory, let alone knowing all the ins and out of every nook and cranny of the TADS 3 library and language; there's probably no one in the known universe who has *that* kind of knowledge. Knowing what you need to know is rather knowing enough to carry out the tasks you commonly carry out in TADS 3 with ease, gradually becoming familiar with the not-quite-so common features as you get to use them more, and knowing where to look up the rest as and when you need it.

Even if you've managed to master the contents of this manual pretty thoroughly, sooner rather than later you'll come up against something you want to do that it doesn't cover. Just what this is will probably be different for every reader; it's what your game does that *isn't* typical that's likely to make it interesting, and no manual can hope to cover all the *atypical* things game authors might like to do with TADS 3.

One place to look if you're stuck on something fairly common is the article on "Where Messages Come From" in the *TADS 3 Technical Manual*; this provides a short sample transcript with notes explaining where the various parts of the transcript are generated, often with links to more detailed documentation. One kind of problem that can be especially frustrating is customizing or getting rid of bits of text that the library displays from some less than immediately obvious place; the article on "Banishing Awkward Messages" also from the *Technical Manual*, provides help with the most common cases.

It may be, however, that your problem is of a very different kind. Depending on the nature of what it is you're trying to do, you should familiarize yourself with the parts of the *System Manual* and *Technical Manual* we haven't really touched on here. If you think that what you're up against hasn't been explained properly anywhere and is something other people might also like guidance on, consider submitting a request to the *TADS 3 Technical Manual* Wish List (at http://www.tads.org/t3techlist.htm) (as well, no doubt, as asking about your difficulty on rec-arts.int.fiction).

At some point, again probably sooner rather than later, you are going to find that there is going to be no substitute for delving deep into the *TADS 3 Library Reference Manual* and trying to puzzle things out for yourself. Although we've tried to cover most of the most commonly used properties and methods of the most commonly used classes, it's not possible to try to cover everything here without making this manual so vast and dense that nobody could ever read it (or write it!). In any case, as soon as you start moving beyond the basics it's probably a good idea to start browsing the *Library Reference Manual* to read more about the classes, objects and functions you may be interested in. Almost however long you've been working with TADS 3 you'll probably discover something new!

If you have a multi-tabbed web browser, it's a good idea to have the main parts of the TADS 3 documentation set (*System Manual*, *Technical Manual* and *Library Reference Manual*) open in separate tabs whenever you start working with TADS 3, so that they're instantly available when you need to look things up (which you will need to do – frequently). Indeed, it's often useful to have the *Library Reference Manual* open in several tabs at once for when you want to jump round the library while keeping track of where you've jumped from.

Another technique you can use if you're using Workbench is to set break-points in the debugger and step through the code to see what's happening, though at some points this may well feel more confusing than helpful, especially at first! The reasonably detailed account of what happens when in the article on "The Command Execution Cycle" in the *Technical Manual* may be of some help here, at least in suggesting where a break point might usefully be set to track down whatever it is you're looking for.

One final piece of advice: if you come up against something that you're absolutely stuck on, go away and try something else. Whether you're still learning TADS 3 or reasonably adept at it, you'll occasionally come up against things that seem just too hard or too complicated, but which later yield to renewed efforts once you come back to them in the light of greater experience.

This manual has taken about you as far as an introductory manual can. From now on the rest is up to you. In the meanwhile, if you find any inaccuracies, typos, glaring omissions, or places where things are unclear or could be improved, do let me know so that I can try to put them right for a future edition. I can be emailed on eric.eve@hmc.ox.ac.uk or eric.eve@ukf.net.

Eric Eve

Harris Manchester College, Oxford

August 2008

21 Alphabetical Index