Learning TADS 3
with adv3Lite

by Eric Eve

Table of Contents

1 Introduction

1.1 The Aim and Purpose of this Book

The aim of this book is to enable people to learn TADS 3 from scratch, but to learn it in conjunction with adv3Lite rather than the adv3 library that comes standard with TADS 3. The reason you may want to do this is that adv3Lite is intended to be easier to learn and work with than adv3. But if you start out knowing little or nothing at all about TADS 3, then you need to pick up quite a bit about the TADS 3 *language* (and other aspects of the system) along with learning how to use a *library*. *Learning TADS 3* does this job for people who want to learn TADS 3 with the built-in adv3 Library; *Learning TADS 3 with adv3Lite* aims to do the same job for people new to TADS 3 who'd rather learn it with adv3Lite.

Adv3Lite already comes with a Tutorial, called the *adv3Lite Tutorial*. While *Learning TADS 3* is still a tutorial, unlike the *adv3Lite Tutorial* it is not tied to taking the reader through the development of a few sample games, and is thus free to present the material in a more systematic manner. It should 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 the *Tutorial* 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 developing your own IF masterpiece, *Learning TADS 3 with adv3Lite* 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 adv3Lite's capabilities, then you may prefer the *Tutorial*.

Learning TADS 3 with adv3Lite is intended as an *introduction* to both the TADS 3 language and the adv3Lite library; it is not intended to tell you everything there is to know about either, since most readers would probably find that overwhelming. The aim is rather to help beginners master the basics, to the point at which they can comfortably find out anything else they know from the *TADS 3 System Manual* and the *Adv3Lite Manual*. For that reason some details are deliberately left out to help the reader focus on the more basic essentials without being distracted into every nook, cranny and by-way. The penultimate chapter then gives a brief account of some of the other features of the system not covered in the book, to alert readers to some of the other things that TADS 3 and adv3Lite can do. It may be, then, that you won't immediately find the answer to how to implement the terribly clever idea you've come up with for your game. On the assumption that you're a beginner, I would urge you to put that terribly clever idea to one side until you are reasonably familiar with the basics set out in this book; you'll get where you want to go a lot quicker that way in the long run.

Becoming proficient at TADS 3 (and adv3Lite) is not a matter of committing everything you might possibly want to know about it to memory. The system is far too large for that. Becoming proficient at TADS 3 is a matter of learning (through practice) those parts of the system that you use 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 *Adv3Lite Manual* are essential skills for any adv3Lite 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 from time to time 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 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 set it to music or translate it into epic Greek hexameters they are entirely welcome to do so! 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, which you can follow up 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, 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. If you do try the exercises, you might like to know that suggested solutions to some of them are provided in the same directory as this book.

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. But you may need instructions on how to set up and compile a game in adv3Lite.

1.2.1   Getting What You Need

The first thing you need is the latest version of the TADS 3 authoring system. To use adv3Lite you need TADS 3.1.2, and preferably TADS 3.1.3 (or higher). If you have an earlier version of TADS 3 than version 3.1.3 please update to the latest version of TADS 3 now, before attempting to do anything else suggested in this book.

For a Windows system, you want the Windows authoring kit that comes with Windows Workbench. For Mac O/S X or Linux/Unix the recommended TADS authoring system is FrobTADS. Both of these can be downloaded from http://www.tads.org/tads3.htm. Even if you don't have a Windows system, however, you might want to consider trying to run Windows Workbench under WINE, Parallels or Crossover (see, for example, the discussion about this on int-fiction.org), since using Workbench gives you an integrated development environment (IDE) that makes many TADS 3 authoring tasks a whole lot easier, although I understand that running Workbench on a non-Windows system like this can be problematic. If you're using a non-Windows system and you don't want to use Workbench (or you can't get it to work) you'll also need to download a text editor for writing your code; Jim Aikin recommends TextWrangler for MacOS (it's free).

The other thing you'll need (obviously) is the adv3Lite library. The fact that you're reading this probably means that you've obtained a copy of it already, but in case you haven't (maybe you're reading this on-line or on someone else's machine), you can obtain the latest version either from https://dl.dropboxusercontent.com/u/58348218/adv3Lite/adv3Lite10.zip or from the IF-Archive.

The adv3Lite library comes in a zip archive containing an adv3Lite folder/directory and a number of subfolders. A good place to unzip this into is the extensions folder/directory under the folder/directory in which you keep, or plan to keep, your TADS 3 game source code. On a Windows system this will typically be under My Documents\TADS 3; the TADS 3 author's kit setup program should create both this folder and the extensions folder under it. On a non-Windows system you may need to create these directories manually yourself.

You should therefore end up with all the adv3Lite material in a folder somewhere like ...\My Documents\extensions\adv3Lite (where ... is the path to your My Documents folder or its equivalent on a non-Windows system). If you need to reinstall adv3Lite (for example, because you want to update to a newer version) I recommend that you first delete the existing extensions\adv3Lite folder/directory before unzipping

the new adv3Lite folder to the same location; this should avoid any problems that might arise from remnants of an older version interfering with the newer version.

1.2.2   Setting It All Up

Once you've got up-to-date versions of the TADS 3 authoring system/compiler and the adv3Lite library you'll need to set things up so they all work together and you can set about creating your game. In what follows we'll assume your game is called "Heidi", since that's the name of the first game in the *Tutorial*. Obviously, when you come to create other games, you can substitute your own name for "Heidi" in what follows.

Set-up for Windows Workbench (with TADS 3.1.3 or higher)

If you haven't already got version 3.1.3 (or higher) of TADS 3, you should consider downloading and installing it before you proceed. This will make it much easier to create new games with adv3Lite.

Assuming you have got TADS 3.1.3 (or higher) and that you've installed your adv3Lite folder under ..\My Documents\TADS 3\extensions as suggested in the previous section, creating a new adv3Lite game is very simple:

1. From the Windows Workbench menu select File -> New Project
2. In the dialog that should then appear, simply click **Next** to continue to the second page of the Wizard.
3. In the second page of the Dialog, type "Heidi" (without the quotation marks) in the upper (**Project Name**) box, then click in the lower(**Folder location**) box. From there you can select an existing folder, but for a new project you should create a new one, so click the **Make New Folder** button and then enter a name for your new folder, such as heidi (making sure you create it under the ..\My Documents\TADS 3\ folder). Then click **OK** followed by **Next**.
4. In the third page of the Dialog, scroll down the list of Project Types until you get to **Adv3Lite**. Click on Adv3Lite to select it and then click **Next**.
5. Fill in the information on the final (Bibliography) page of the wizard (e.g. with the name Heidi, your own name and email address, and a brief description like "A small tutorial game") and then click **Next** to complete the wizard. Your new adv3Lite game (or at least the beginnings of it) will then be created for you.

Set-up for Windows Workbench (with earlier versions of TADS 3)

If for any reason you have an earlier version of Windows Workbench and it isn't convenient to upgrade, you will need to go through the following steps:

First, before attempting to set up an individual game, open Windows Workbench, Select Tools from the Menu, then Options. Click on "Library Paths" (about three-quarters of the way down on the left-hand pane) then click on the "Add folders..."

button. Navigate to the folder where you installed adv3Lite in the previous section (typically something like C:\Users\YourName\Documents\TADS 3\extensions\adv3Lite) and select it to add it to the list. This will tell Workbench where it can find the adv3Lite library; once you've done this once you shouldn't have to do it again (unless perhaps you reinstall Workbench for any reason).

Now, to set everything up to create a new game (which we're calling "Heidi", although the same principles apply whatever you're calling it) carry out the following steps:

1. Under your My Documents\TADS 3 folder create a new folder called Heidi.
2. Locate the extensions\adv3Lite\template folder and copy (don't move) its entire contents (but not the folder itself) into your newly-created Heidi folder.
3. Navigate back to the Heidi folder.
4. In the Heidi folder, rename the file **adv3Ltemplate.t3m** to **heidi.t3m**.
5. In the Heidi folder, rename the file **adv3Ltemplate.tdbconfig** to **heidi.tdbconfig**.
6. Open the file heidi.t3m in Workbench (either by double-clicking on it in Windows Explorer, or by using the File -> Open Project option from the menu in Workbench).
7. Click the Go button (the little blue triangle at the left-hand end of the toolbar) in Workbench to compile and run the file to make sure everything's okay. You should then be in a position to start working on your new adv3Lite project.

Set-up for Mac OS X or Linux/Unix

To set everything up to create a new game (which we're calling "Heidi", although the same principles apply whatever you're calling it) carry out the following steps:

1. This assumes you've placed the adv3Lite directory under an extensions directory under your TADS directory, and that you'll create your Heidi directory in the next step under the same TADS directory.
2. Under the directory you've created to hold your TADS 3 source code (you might have called it TADS 3) create a new directory called Heidi.
3. Locate adv3Lite/template directory and copy (don't move) its entire contents (but not the directory itself) into your newly-created Heidi directory.
4. Navigate back to the Heidi directory.
5. In the Heidi directory, rename the file **adv3Ltemplate.t3m** to **heidi.t3m**.
6. In the Heidi directory, delete the file **adv3Ltemplate.tdbconfig** (if you're not using Workbench, you don't need it).
7. Now open the heidi.t3m file in a text editor and edit it to read:

```
-D LANGUAGE=english
-Fy obj -Fo obj
-o heidi.t3
-lib system
-lib ../extensions/adv3Lite/adv3Lite
-source start
```

You can delete any instances of comments like the "warning — this file was mechanically generated" paragraph you find in the t3m file, together with any bits of executable code you find there. You can also delete the line -pre. You may need to change the penultimate line if the path to where you've stored the adv3Lite directory is different from that assumed here.

8. Open a Terminal window. The Terminal program is located in Applications > Utilities. You may want to make an alias for it and drag it into your Dock.

9. In the Terminal, use the cd (change directory) command to navigate to the folder where your game files are stored. For instance, you might type 'cd Documents/TADS/Heidi' and then hit Return.

10. While the Terminal is logged into this directory, you can compile your game using this command:

```
t3make -d -f heidi
```

If all goes well, you should see a string of messages in the Terminal window, and a new file (heidi.t3) will appear in the Heidi directory. This is your compiled game file. If you've installed an interpreter program that can run TADS games, you'll be able to double-click the .t3 file and launch the game to test your work. Alternatively, you can run the game directly in the Terminal by typing 'frob heidi.t3' and hitting Return.

11. Keep the Terminal window open and press the Up arrow on the keyboard each time you want to do a compile, as this will reload the last command line that you typed (t3make etc.).

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 <tads.h>
#include <advlite.h>

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

gameMain: GameMainDef
    initialPlayerChar = me
;

me: Thing
    isFixed = true
    person = 2  // change to 1 for a first-person game
    contType = Carrier
    location = startRoom
;
```

```
startRoom: Room
   roomTitle = 'Starting Room'
   desc = "This is the starting room. "
;
```

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 adv3Lite game in order to start with the skeletal code shown above. If you're not using Workbench you can instead copy the necessary files from the template directory of your adv3Lite installation; copy the contents of the template directory to your working directory, and rename the files to something else (mygame.t, for example) and you can use them as the basis for your own exercises.

1.3   Feedback and Acknowledgements

Special thanks are due to Jim Aikin, Mark Engelberg and Knight Errant for pointing out various errors in the adv3 version 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
    roomTitle = '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: roomTitle (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 adv3Lite are the eight compass directions: **north**, **south**, **east**, **west**, **northeast**, **northwest**, **southeast**, **southwest**, together with the four special directions: **up**, **down**, **in** and **out,** and the four shipboard 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 adv3Lite 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 those 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

`roomTitle` 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 *Adv3Lite Library Reference Manual* (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
'roomTitle' 'vocab' "desc"?;


Room
'roomTitle' "desc"?;
```

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 `roomTitle` property. If there's a second item in single-quote marks it will be the `vocab.` Whether we define the Room with one or two 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' 'auditorium'
    "The auditorium is thronged with a great press of people. "
;
```

Then 'Auditorium of Albert Hall' would the `roomTitle`, 'auditorium' would be the `vocab`, and "The auditorium is thronged with a great press of people. " would be the `desc`.

After a while, the common templates (e.g. for `Room`) quickly become familiar, and you shouldn'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.

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   Rooms and Regions

So far we've only met one kind of room, defined with the `Room` class. In adv3Lite, that's the only kind of Room there is.

We can, however, do several things to customize a room. For example, by default rooms all start out lit, but it we want a dark room we can simply define it isLit property as nil:

```
cellar: Room 'Cellar'
   "If you could see anything in here it would all look rather dusty."
    isLit = nil
;
```

But in fact, unless the player character has a light source, you won't see anything in there, so you'll just get a message saying it's too dark to see anything. We'll return to that in Chapter 10 when we come to consider Darkness and Light.

If you had a whole lot of dark rooms in your game you might want to define a DarkRoom class like so:

```
class DarkRoom: Room
   isLit = nil
;
```

Then you could make definitions like:

```
cellar: DarkRoom 'Cellar'
   "A dusty passage leads off to the east. "
   east = dustyPassage
;

dustyPassage: DarkRoom 'Dusty Passage'
    "This passage seems to run forever from west to east. "
   west = cellar
   east = self
;
```

This is known as making use of *inheritance* which we'll talk about in more detail below.

Something else we can do with a room is to put it in a *Region*. A Region is a set of rooms that belong together in some way, like all the rooms in a house, or all the rooms on the ground floor, or all the rooms outdoors. A room can belong to more than one region, and regions can belong to other regions, so that, for example, we could

have a hall that's in the downstairs region of a myHome region, thus:

```
downstairs: Region
   regions = [myHome]
;

myHome: Region
;

hall: Room 'Hall'
    "The front door lies just to the north..."
    regions = [downstairs]
;
```

This puts the hall in both the downstairs region and the myHome region. Notice how we define what regions a room or region is in: we list those regions in the **regions** property. Anything in square brackets [] denotes a *list*, the elements of which must be separated by commas. Don't worry about that too much for the moment, however; we'll talk about lists in more detail in a later chapter.

You may be wondering what Regions are actually good for. They can in fact be used for a variety of purposes. We can test whether something or someone is in a region; we can make things happen when someone leaves or enters a region; or we can restrict what happens according to region (so that, for example, we could restrict the use of the shipboard directions port, starboard, fore and aft to a shipboard region). But how to do all this depends on concepts we haven't met yet.

At this point, we'll introduce another property you can use with rooms: **cannotGoThatWayMsg.** This is the message that would be displayed if the player tried to go in any direction we haven't explicitly defined an exit for. For example, we might expand our definition of the hall a bit like this:

```
hall: Room 'Hall'
    "The front door lies just to the north. Other exits lead east and west, while
      a flight of stairs runs up to the south. "
      regions = [downstairs]

    north = drive
    south = landing
    east = livingRoom
    west = kitchen

    cannotGoThatWayMsg = 'You\'d just end up in a corner! '

;
```

Note how we use the backslash (\) immediately before single-quote marks that we want to appear within a single-quoted string property. An alternative would have been to use three single-quote marks to start and end the string value:

```
 cannotGoThatWayMsg = '''You'd just end up in a corner! '''
```

This avoids the need to type the awkward backslash (\) character before the single

quote marks (or apostrophes) that appear in the body of the string, but the meaning is otherwise identical. (If you find it confusing to have different ways of doing the same thing, then you can always ignore this alternative for now).

Exercise 2: Now that we've learned a bit more about rooms, try to construct a more adventurous map using examples of the new properties we've just seen. This might, for example, include the inside of a house and part of its garden. Even though we haven't really discussed what you can do with Regions much yet, try putting each room in an appropriate region.

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, our custom-made `DarkRoom` class inherited from `Room`. Although we haven't met either class yet, `Room` in turn inherits from both `TravelConnector` and `Thing`. Inheriting from more than one class is called *multiple inheritance*.

Classes are extremely useful when we want to define 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 be light or dark; 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, we might want to make another kind of room (although this may not often be necessary with adv3Lite). An example we gave before was that of defining a `DarkRoom` class for a room without any light:

```
class DarkRoom: Room
   isLit = nil
;
```

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.

Since Rooms aren't the only kind of object that have the `isLit` property, we could have gone about this a different way by defining a `Dark` class, like this:

```
Dark: object
   isLit = nil
;
```

We could then have defined our DarkRoom class using multiple inheritance, like this:

```
class DarkRoom: Dark, Room
;
```

To be sure this is such a trivial example that it's unlikely that anyone would ever define this particular set of classes in practice, but it serves to illustrate the principle, and the principle is this: 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 **Dark** 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 System Manual*.

2.5   A Bit More About 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 use a special kind of *embedded expression*, which is something inside a string enclosed in **<< >>** (we'll say more about these in general later). Two varieties of embedded expressions that can be useful in room descriptions are **<<first time>>...<<only>>** and **<<one of>>** … **<<or>>** … **<<stopping>>**. For example, suppose you have a room that's initially described as "The first thing you notice about this hall is that it continues to the east" and subsequently as "This large hall continues to the east." We could use the **<<one of>>** … **<<or>>** … **<<stopping>>** to deal with this for us, thus:

```
hallWest: Room 'Hall (west)'
    "<<one of>>The first thing you notice about this hall is that it <<or>>This
      large hall<<stopping>> continues to the east. "
    east = hallEast
;
```

There's just a couple more points we'll make about rooms for now before we move onto other things. A name like 'Hall (west)' is fine for the display name of a room in the status line or as the heading of a room description, but it doesn't work quite so well as a name for the room in other contexts, such as "You see Richard over in Hall (west)" or "A walking stick lies abandoned on the ground in Hall (west)." In such cases it would be more natural to refer to the room as something like 'the west end of the hall'. We can do this by defining the room's **vocab** property, which incidentally sets its **name** property at the same time (don't worry, we'll come to a fuller explanation of that when we come to discuss Things in the next chapter). For a room, we can achieve that simply by using a second single-quoted string in the Room template, like this:

```
hallWest: Room 'Hall (west)' 'west end of the hall'
    "<<one of>>The first thing you notice about this hall is that it <<or>>This
      large hall<<stopping>> continues to the east. "
    east = hallEast
;
```

This actually achieves two things: first, it ensures that the game will use the name 'west end of the hall' if and when it ever needs to construct a sentence about this Room, and second, it means that the player can refer to this room as 'west end of the hall' in commands like GO TO WEST END OF HALL.

Much of the time, though, we'll be defining rooms where the **roomTitle**, name and **vocab** are effectively all the same, rooms with name like 'kitchen', 'dining room', or 'large meadow'. In such a case adv3Lite will take both the **vocab** and the name to be the **roomTitle** in lower case (e.g. a **roomTitle** of 'Kitchen' becomes a name and **vocab** of 'kitchen'). This takes place for any room that doesn't explicitly define a **vocab** property of its own, provided the room's **autoName** property is set to true (as it is by default). If you don't want this behaviour, you can can set **autoName** to **nil** (meaning false in this context). A further complication is that you may have a room name (i.e. **roomTitle**) like 'West Street', that's effectively a proper name, i.e. one that shouldn't be turned into lower case when converted into the name property, and one that shouldn't be preceded by articles like 'the' or 'an' when it appears in a sentence ('You last saw Martin in West Street', not, 'You last saw Martin in the west street'). In such a case, we need to define **proper = true** on the room, like so:

```
westStreet: Room 'West Street'
    "West Street runs straight east-west along a row of old-fashioned shop
      fronts..."
    proper = true
;
```

There's one last thing to say about rooms at this stage. You'll have seen that we can define shipboard directions (like port, starboard, fore and aft) on any room, but on most rooms they probably won't be meaningful, and in a game where there are no rooms aboard a ship or comparable vessel, they may never be meaningful. In such a case the default responses to commands like **go port** or **push trolley aft** or **throw knife starboard** may seem less than fully appropriate; what's really needed in such cases is a message saying that these directions are meaningless in this context.

You can control which directions are considered meaningful on any given room by using the properties/methods `allowShipboardDirections` and `allowCompassDirections`. If one or the other of these is nil the command will be stopped in its tracks with a message like "Shipboard/compass directions have no meaning here." You might want to block compass directions, for example, for rooms that are meant to be aboard a ship.

In the library `allowCompassDirections` is simply defined as true, leaving you to set it to nil if and where you deem it appropriate on particular rooms. It's a bit different with `allowShipboardDirections`, however. In this case, the game looks to see if any of the shipboard directions (port, starboard, fore or aft) have been defined on the room in question; if they have, then `allowShipboardDirections` will be true; otherwise it will be nil. For the most part this means you can just leave `allowShipboardDirections` to look after itself, but there may occasionally be rooms where you might want to override it. For example, if a room represents the hold of a ship from which the only way out is up, you may want to make `allowShipboardDirections` true on the grounds that the shipboard directions are still meaningful in the hold (we know which way port, starboard, fore and aft are) even if we can't actually travel in any of them; in such a case you might want to do something like this:

```
hold: Room 'Hold'
    "The ship's hold is dark and dank, with water seeping in between
     the planking. The only way out is back up to the main deck. "
     up = mainDeck
     allowShipboardDirections = true
;
```

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, 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
    vocab = 'red ball; small red hard round cricket'
    location = frontLawn
    desc = "It's quite small and hard; it looks much like a cricket ball. "
;
```

Since these three 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 'red ball; small red hard round cricket'  @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 adv3Lite game. Even if you never get round to learning any other adv3Lite 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.

vocab defines two things: the short name of the object (in this case 'red ball') and the words the player can use to refer to the object in commands. In nearly every case the former (the short name) will be subset of the latter (the words the player can use to refer to the object), which is why we define both together in the same property. The short name of the 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." We start the **vocab** property by giving the short name followed by a semicolon. Then we list all the other adjectives that might be used to refer to the object. If there were any other nouns, we'd then have another semicolon followed by the list of the other nouns. We could also have a third semicolon followed by the pronoun used to the refer to the object, for example:

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

If we don't specify a pronoun the game will assume it's 'it', so we only need to specify a pronoun if it's 'him', 'her' or 'them'.

The **desc** property defines the description of the object that is displayed when the object is examined. Note that unlike the **vocab** property, 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 ball; small red hard round cricket' '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.

We should next look at the **vocab** property in a bit more detail. While, in a sense, it's one property – we can define it as a one single-quoted string – in another sense it

provides a means of defining several properties, making it a compact and efficient tool for defining several of an object's most commonly-needed properties all together at once. The basic structure of the `vocab` property divides into four sections, each divided from the next by a semicolon, which may be schematically represented like so:

```
'name; adj adj adj; noun noun noun; pronoun'
```

That is, the first section of the `vocab` property actually defines the `name` property of the Thing we're defining (this holds good for Rooms too, by the way). It also adds all the words it finds in the name part into the Thing's `vocabWords` property so they can be used to refer to the Thing we're defining. This means (a) that we (virtually) never have to worry about the `vocabWords` property in our own code, and (b) that we don't need to repeat any of the words in the Thing's name in order to define the words that can be used to refer to it.

The second and third sections can then be used to add any additional adjectives and nouns (respectively) that can be used to refer to this thing. For example, if your game includes what American players would call a 'flashlight' and British users a 'torch', it would be good to include both nouns somewhere, as well as any adjectives used it its description, for example:

```
flashlight: Thing 'flashlight; heavy black rubber; torch'
   "It's a heavy black rubber flashlight. "
;
```

The fourth and final section can be used to define the pronoun (or pronouns) that could be used to refer to the object. By default, the library will assume this to be 'it', which is likely to be correct for many if not most of the objects you define. But some will be plural, and some may be masculine or feminine, for example:

```
trousers: Thing 'blue trousers; old dark; pants; them'
   "They're your old dark blue trousers. "
;

oldWoman: Thing 'old woman; handsome;; she'
    "She may be quite old, but she's quite handsome with it. "
;
```

The second of these definitions incidentally illustrates that we can leave a section empty if we don't need it; we just have a couple of semicolons together after the adjective.

Defining the pronoun at the end of the `vocab` property is actually doing rather more than just defining what pronoun can be used; it's in effect defining the gender and number of the object when it's other than the default neuter singular (of course you could do that too by defining an explicit 'it' here, but there's no need to do that in practice). We could define these properties explicitly by using the `isHim`, `isHer` and

plural properties of Thing, but (once you get used to the idea) it's quicker and easier simply to add the appropriate pronoun at the end of the **vocab** property.

The trousers/pants example raises another possibility, however, since we might want to call this object a 'pair of trousers' (or 'pair of pants'). Is this then singular or plural? The initial answer is determined by what would be the correct form of the verb 'to be' following the name of this object in a sentence. In other words, would we say "The pair of trousers is blue" or "The pair of trousers are blue"? Most people would probably regard the former rather than the latter as correct, so the trousers need to be singular, 'it' rather than 'them'. On the other hand the player might well refer to the pants/trousers as 'them'. We might say, then, that such an object is *ambiguously plural*, and we can indeed declare it as such by setting its **ambiguouslyPlural** property to **true**. But once again we don't need to do this explicitly; we can do it implicitly by including both pronouns in the **vocab** property of the trousers object:

```
trousers: Thing 'pair of trousers; old dark blue; pants; it them'
    "They're your old dark blue trousers. "
;
```

In this case the order of the pronouns is significant: 'it' comes first because the pair of trousers has to be treated as grammatically singular when it forms the subject of a sentence. Putting 'them' second signals that we're regarding the pair of trousers as grammatically singular, but ambiguously plural insofar as the player can also refer to the trousers with the pronoun 'them', in such interchanges as:

>X TROUSERS
They're your old dark blue trousers.

>TAKE THEM
Taken.

Up until now we've mentioned that there are adjective and noun sections to the **vocab** property without really explaining the significance of the distinction. In adv3Lite the parser doesn't care about the order of words used to match an object (unless you explicitly want it to, but that's a topic we'll leave till much later), but it does care about parts of speech. In a nutshell, it ignores articles (words like 'a', 'an', 'some' and 'the') treats prepositions ('to', 'from', 'with', 'of' or 'for') as a special case (see below) and prefers nouns to adjectives. That means if we have one object called 'orange bowl' and another called 'orange' (i.e. a piece of fruit), a command like TAKE ORANGE will be directed to the fruit, not the bowl, since in the case of the fruit 'orange' is a noun, while in the case of the bowl it's only an adjective.

How that works in the adjectives and noun sections of the **vocab** property should be reasonably obvious. Less obvious is how the library determines the parts of speech of the words in the name section. What happens is that the library assumes that every

word except the last is an adjective, and that the final word is a noun. If, however, the library encounters a preposition in the name (e.g. 'large piece of rich chocolate cake') it assumes that the words before the preposition constitute a noun phrase ending in a noun possibly preceded by one or more adjectives (in this instance, for example, 'large piece', so that 'piece' would be treated as a noun, while 'large', 'rich', 'chocolate' and 'cake' would be treated as adjectives; 'cake' could be made a noun by immediately following it with '[n]'). Any article immediately following a preposition is simply ignored (i.e. not entered into the object's vocabulary), so that, for example, with a name like 'west end of the hall[n]', 'end' and 'hall' will be entered as nouns, 'of' as a preposition and 'the' ignored. (The reason the library treats every word in the name following a preposition as an adjective, even if it looks like a noun, is that in phrases like 'bottle of wine' and 'key to the garage', 'wine' and 'garage' effectively act as adjectives qualifying 'bottle' and 'key', so they are considered as less important to the match; in phrases like 'pair of shoes' or 'flight of stairs' the library's assumption is less obviously appropriate, so in such cases we'd probably want to follow 'shoes' or 'stairs' with '[n]' to force the library to treat it as a noun).

We said above that the library treats prepositions as a special case. What that means is that a preposition must be entered into an object's vocabulary for it to be matched by player's input, but that the preposition won't match by itself. Thus, for example, an object given a **vocab** property of 'piece of cake' will indeed match a command like EAT PIECE OF CAKE, but won't match EAT OF.

There may occasionally be other words we want treated like this, for example if 'his' or 'her' or 'your' or 'my' figured in the name of an object, we probably wouldn't want to match it on such a word alone. In such a case we can mark the word as a weak token, either by following it with [weak] or by enclosing it in parentheses; the two are equivalent:

```
tHerName: Topic 'her[weak] name';
tHerName: Topic '(her) name';
```

We haven't encountered Topics yet, so don't worry too much about what they are just now; the examples nonetheless serve to illustrate the principle that either way of defining this Topic would allow it to match 'her name' but not just 'her'.

We can use an analogous technique to override the library's default assumptions about parts of speech by following a word with [n] to make it a noun, [adj] to make it an adjective, [prep] to make it a preposition, or [pl] to make it a plural. For example, if we had a character called John Smith we might define him thus:

```
john: Thing 'John[n] Smith'; tall; man; him'
;
```

Likewise, if we used the first definition of our trousers object, but wanted to allow the player to refer to it as a pair of trousers as well, we'd list 'of' as an adjective but follow it by [prep]:

```
trousers: Thing 'blue trousers; old dark of[prep]; pants pair; them it'
    "They're your old dark blue trousers. "
;
```

The John Smith example brings us to another point. Normally, when the parser needs to construct a sentence using the name you've given it, it will add the appropriate article, e.g. "You are carrying a book and an apple" or "You can't eat the book." In the case of something like John Smith that's clearly inappropriate: we don't refer to him as 'a John Smith' or 'the John Smith' (except perhaps when other people of the same name may be in view), but normally just as 'John Smith'. In other words, we don't generally use an article with something that has a proper name. We could deal with the John Smith example by explicitly defining `proper = true` on the `john` object, but it turns out that we don't actually need to. When all the words in a name start with a capital letter, adv3Lite will assume it's a proper name without our having to say so explicitly.

In other cases we may occasionally need to tell the library what article to use. For example, a mass noun like 'snow' should be described as 'some snow' not 'a snow', so we need to include the article 'some' explicitly at the start of the name:

```
snow: Thing 'some snow; crisp white'
;
```

This, incidentally, has the effect of setting the snow's `massNoun` property to true, once again something we don't need to do explicitly.

Occasionally, we may come across a qualified noun, that is one which doesn't strictly speaking have a proper name but still shouldn't take an article. An example might be an object called "Jill's bag"; the library won't treat this as a proper name since not every word in the name starts with a capital letter, but we can make it treat it as a qualified name by starting the name with (), thus:

```
jillBag: Thing '() Jill\'s bag'
;
```

Note, by the way, the use of the backslash (\) to 'escape' the apostrophe (') in 'Jill\'s bag'; we escape the apostrophe that way to tell the compiler that this ' isn't the closing quote-mark of the single-quoted string.

We've said nothing so far about plurals. In the main we don't need to worry about them, since the library can take care of them for us automatically. Suppose, for example, we define two books thus:

```
redBook: Thing 'red book'
;

blueBook: Thing 'blue book'
;
```

The library will be quite capable of working out that 'books' is the plural of 'book' (and it can do this with a lot of irregular plurals too (such as 'men' and 'feet'). So the game will understand perfectly well if the player types TAKE BOOKS:

>TAKE BOOKS
You take the red book and the blue book.

It can also create plurals for itself in text it outputs. For example, suppose you were to define the following three objects in your game:

```
coin1: Thing 'gold coin' @hall
;

coin2: Thing 'gold coin' @hall
;

coin3: Thing 'gold coin' @hall
;
```

Then, when the player looks around in the hall, the game will say "You see three gold coins here." In other words, without the game author having to do any more about it the game recognizes that the three coins have identical names, so it groups them together and describes them in the plural.

In sum, then, the **vocab** property provides a neat and efficient way of simultaneously defining (or potentially defining) a number of different properties of Thing, namely **name**, **vocabWords**, **proper**, **massNoun**, **qualified**, **plural**, **ambiguouslyPlural**, **isHim**, **isHer**, **isIt**. These in turn help to define several dependent properties such as **aName** and **theName**, which give the name of the object preceded by the appropriate form of the indefinite or definite article respectively (or no article at all if the Thing is **proper** or **qualified**). This may seem rather a lot to take in all at once, but it quickly starts to become familiar once you start using it, and it's worth becoming familiar with it, since mastering the proper use of the **vocab** property can save you a lot of time and effort in the long run.

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 as you'll find if you experiment, there's not much you can do with them. You can pick them up, carry them around, put them down again and throw them at other things, and that's about it. We can gain a little more variety in what Things can be do by overriding a few simple properties and/or using using 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. Defining an object to be of class **Wearable** is the same as defining it as a **Thing** and then defining **isWearable = true** on it.

- **Food** – something the player character can eat. Defining an object of class **Food** is equivalent to defining it as a **Thing** and defining **isEdible = true** on it.

- **Switch** – something the player can turn on and off. Defining an object to be of class Switch is equivalent to defining a Thing with **isSwitchable = true**.

- **Flashlight** – a portable light source than can be turned on and off (this is a special kind of **Switch** that becomes lit when on). 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.

We can also extend the behaviour of a Thing by manipulating some other basic properties. For example:

- **readDesc** – if this is defined (if used it should be defined as a double-quoted string) **readDesc** provides the response to READ SOMETHING.

- **isHidden** – if this is **true** then the object is hidden from view (even if it would otherwise be in plain sight).

- **IsLit** - is this is true them the object will act as a light source.

These may become clearer with a couple of examples. Suppose, for example, we have an object representing a newspaper, and we want reading it to provide a different response from merely examining it. We could define it like this:

```
newspaper: Thing 'newspaper; daily of[prep]; copy paper diatribe' @hall
    "It's a copy of the <i>Daily Diatribe</i>. "

    readDesc = "You find it contains the usual collection of depressing
        articles detailing how the world is going even further to the dogs
        than even the most determined pessimist hitherto thought possible. "
;
```

Note how we can use simple HTML mark-up like **<i>...</i>** to format the text (here putting *Daily Diatribe* in italics).

One use of **isLit** can be illustrated from a possible way of defining the **Flashlight** class:

```
class Flashlight: Switch
   isLit = isOn
;
```

In fact, the library's definition of Flashlight is a bit more complicated than that, using a number of features we haven't encountered yet.

It's a bit difficult to illustrate the use of isHidden without using features of the language we haven't encountered yet, but since they're about to come up in the next coding excursus, perhaps we may anticipate them here. Suppose that when we examine a desk we find it has a secret knob underneath that we hadn't noticed before. At a first approximation we could do something like this:

```
desk: Thing 'desk; plain wooden' @study
   desc()
   {
       "It's just a plain wooden desk. ";
        if(knob.isHidden)
        {
            "Closer examination, however, reveals that it has a secret knob
            half-concealed on its underside. ";
            knob.isHidden = nil;
        }
   }
;

knob: Thing 'secret knob' @desk
   isHidden = true
;
```

Note how we can define **desc()** as a method rather than just a double-quoted string, but that if we do so we have to define it explicitly (with its name) and not through the template. Note also that instead of writing **knob.isHidden = nil;** we could have written **knob.discover();** this would have enabled the definition of the desk to be written as:

```
desk: Thing 'desk; plain wooden' @study
   "It's just a plain wooden desk. <<if knob.isHidden>> Closer examination,
    however, reveals that is has a secret knob half-concealed underneath.
    <<knob.discover()>> <<end>> "
```

```
;
```

Although whether that should be considered an improvement is perhaps a matter of taste, since many people might find the first form clearer. There are in any cases problems with either method, since there's nothing to stop the player from picking up the desk and carrying it around, or indeed the knob. We shall see how to prevent such things in the final section of the chapter. In later chapters we'll see other ways to hide things.

Exercise 4: Try adding some `Wearable`, `Food` and `Switchable` objects to your map. Also, add a `Flashlight` which can be used to light up a dark room, and an object with a `readDesc`.

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 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;
```

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`.

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:

```
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 common kinds of statement 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;
```

In the special case where we want to ensure that we assign a non-nil value to something, we can use the if-nil operator `??`. For example, suppose we have:

```
someValue = a ?? b;
```

This will assign the value *b* to `someValue` if *a* is nil, but will otherwise assign *a* to `someValue`. 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 be 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).

To stop things being portable, we can define **isFixed = true** on them. Alternatively, we can define them using one of the classes we're about to meet below. Either way the effect on such objects will be:

- 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 **useInitSpecialDesc** condition). For example, we might define:

```
+ ring: Wearable '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 non-portable object 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'
      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.

The various classes we can use to define non-portable objects are as follows:

- **Fixture** – An object that's obviously fixed in place, like a house or a shelf nailed to the wall.

- **Heavy** – An object 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.

- **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 `notImportantMsg` property.

- `Immovable` – an object that can't in fact be picked up even though this isn't immediately obvious to the player, for example an item that's heavier than it looks. The distinction between this and a `Fixture` is a subtle one: what it boils down to is that the parser will be readier to choose X in response to a TAKE X command if X is an `Immovable` than if it's a `Fixture`.

- `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`.

If the player tries to take any of these kinds of object, or indeed to put them somewhere else or move them, the default response will tend to be a rather bland "The whatever is fixed in place" (or the equivalent appropriate to the class in question). We can customize these responses by overriding one or more of the following 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 the object in question. By default we just use the `cannotTakeMsg` (so any changes to the `cannotTakeMsg` will be copied to the `cannotMoveMsg`).

- `cannotPutMsg` – a message (again typically a singe-quoted string) shown in response to put the object in question in, on, under or behind something. Again, by default, we just use the `cannotTakeMsg`.

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

```
cabinet: Immovable 'large wooden cabinet; bulky polished of[prep];
    furniture piece'
```

```
    "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. '
;
```

Note that because all these objects are ultimately subclasses of Thing, we can (and usually will) use the `Thing` template with them. So, for example, our previous desk with secret knob example could become:

```
desk: Heavy 'desk; plain wooden' @study
    "It's just a plain wooden desk. <<if knob.isHidden>> Closer examination,
     however, reveals that is has a secret knob half-concealed underneath.
     <<knob.discover()>> <<end>> "
;

knob: Fixture 'secret knob' @desk
    isHidden = true
    cannotTakeMsg = 'The knob is firmly attached to the desk. '
;
```

`Unthing`, however, has a special template of its own, since it's generally more useful to define its `notHereMsg` property:

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

In this example 'gold ring' is the `vocab` property 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).

Finally, you should have noticed that we've been defining some properties as single-quoted strings and others as double-quoted strings. The distinction is important and will explained more fully in the next chapter, but for now it is important to remember not to mix the two up, otherwise things won't work properly. As a rough rule of thumb properties with names ending in `desc` need to be defined as double-quoted strings, while those with names ending in `msg` need to be defined as single-quoted strings.

Exercise 6: Look up the `Fixture` class in the *Library Reference Manual* and take a quick look at what classes it inherits from and the list of classes that inherit from it. 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 non-portable object described above.

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 later; for now we'll concentrate on doors as connectors between locations.

A physical door has two sides; in adv3Lite 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 adv3Lite 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 each side to point to the other side. This not only determines where an actor ends up when he or she goes through the door, it also 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 door we use the `Door` class. This inherits from a two classes, namely `Thing` and `TravelConnector` (which we'll say more about later in this chapter). It also has `isFixed = true` by default, so it's not possible for the player to pick up a door and carry it around, and doors are 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 Door class also defines `isOpenable = true`, since doors can normally be opened and closed.

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'
   otherSide = frontDoorOutside
;

drive: Room 'Front Drive'
    "The front door into the house lies to the south. "
    south = frontDoorOutside
;
+ frontDoorOutside: Door 'front door'
    otherSide = 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 `Thing`, we can once again use the `Thing` template to define its `vocab`, and `desc.` Finally, note the use of the plus sign (+) to locate each side of the door in the room in which it belongs.

The doors we defined above will start out closed. If we want them to start out open, we should define `isOpen = true` on both sides of the door (actually it's usually enough to define this on one side of the door, since the library will assume that if one side of a door starts out open, the other side must too, but it may be as well to be explicit in your code).

As mentioned above, a `Door` defines `isOpenable = true`. 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 again 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 object properties (such as `desc` and `specialDesc`) can be 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 effectively a short-hand way of defining a method that displays that string; so for example:

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

is almost 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 adv3Lite 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. (Actually, this is a slight oversimplification, since there are a few situations in which TADS 3 treats a double-quoted string property a little differently from a method, but the above account will serve as a first approximation).

Similarly whenever any 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), at least, not in the obvious way. The following code is illegal:

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

There is a way to change the value of a double-quoted string property; it can be done like this:

```
changeDesc()
{
    setMethod(&desc, 'It\'s a great big heavy ball. ');
}
```

But the need to resort to this kind of thing will probably be rare, especially when you're just starting out in TADS 3, so for the moment we'll just note the possibility and move on.

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,. For example, suppose we wanted to do something with the `desc` property of some object; we can use code like this to recover the single-quoted string equivalent of the `desc` property:

```
    local str = getMethod(&desc);
```

Note the ampersand (&) which is needed before the property name (desc) here; technically speaking `&desc` is a *property pointer.* If that doesn't mean much to you right now, don't worry; just remember you need to use the ampersand before the property name in this kind of situation. Note also that `getMethod()` will only do what you want in this situation if the `desc` property has been defined as a double-quoted string rather than a method (this is one of the few situation in which it makes a difference). If the `desc` property (or whatever property it is you're interested in) might contain a method, you're better off capturing its single-quoted string equivalent with:

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

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). If we want we can even set `desc` to the new value of `str` once we've finished manipulating it:

```
setMethod(&desc, str);
```

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 adv3Lite has classes to model all of these. Unlike doors, these other kinds of connectors don't have to be used in pairs, and instead of defining their `otherSide` property we define their `destination` property to determine which room they lead to. The physical connector classes available in adv3Lite are:

- `StairwayUp` – A `StairwayUp` is something we can climb up. Although we can (and often will) use these for flights of stairs, we can also them for anything that an actor might climb up, such as hillsides, trees and masts.

- `StairwayDown` – Just like a `StairwayUp` except that we can climb down it instead of up it. We may often want to pair a `StairwayUp` with a `StairwayDown` to represent both ends of the same staircase at the lower and upper levels, but there's no necessity to.

- `Passage` – 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.

- `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**.

- `Door` – We've already discussed `Doors` above, but we include them here for the

sake of completeness.

- **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 reveal a secret passage behind. In this case we may want both the name of the object and the vocabulary used to refer to it to change according to whether it's open or closed. We can do that by defining its **vocabWhenOpen** property. This should be defined in exactly the same way as the **vocab** property, but will be applied to the **SecretDoor** when it's opened. If it's closed again its original **vocab** will be restored. To make a door that's completely invisible when closed (for example a concealed panel in a wall), just use an ordinary **Door** and then define **isHidden = !isOpen** and **isConnectorApparent = isOpen** on it.

All these classes are used to represent physical objects at one or 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 'stairs;;; them'
    destination = hall
;

+ junk: Decoration 'pile of useless junk[n]'
   "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'
    "The long passage leads off to the west. "
    destination = kitchen
    getFacets = [kitchenPassage]
;
```

```
+ hallStairs: StairwayDown 'flight of stairs'
  destination = cellar
;

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

+ kitchenPassage: Passage 'long passage'
   destination = hall
   getFacets = [hallPassage]
;
```

There are a few extra points to note here. First, note the use of the `getFacets` property on the hall passage and the kitchen passage. This is a way of indicating that they are the two halves of the same object, namely the passage leading from the kitchen to the hall. This is only ever relevant if the player refers to one end of the passage in a command and then tries to refer to the other end with a pronoun, for example GO THROUGH PASSAGE; X IT. In this case also the 'passage' in the first command might have referred to the hall passage (say), the parser will recognize that IT might refer to the kitchen passage. In the case of the two sides of a Door, the library sets up this getFacets relationship for us; if we want it anywhere else we have to do it ourselves.

Second, 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's one more class that at first sight may look rather like the kind of connector we've just been discussing, but is in fact something a little different. This is the `Enterable` class, which is typically used to represent something like the outside of a building, which a player may attempt to enter with a command like **go into building**. Where the player character then ends up can then be defined either using the `connector` property (which defines the connector, typically a Door, the player must travel via to get inside) or the `destination` property, which simply specifies the room the player will end up in.

We can illustrate all this by means of an example. Suppose we are defining a front drive location which mentions a large house to the south. We'd then use an `Enterable` to represent the outside of the house and point its `connector` property to the front door, something like this:

```
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 'large Georgian house;; building'
      "It has a white-painted front door. "
       connector = frontDoor
;

+ frontDoor: Door 'front door; white painted white-painted'
    "It has been painted white. "
    otherSide = frontDoorInside
;
```

Exercise 9: Now that we've covered both `Passages, Enterables` and the like, 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 stairs, passages and `Enterable` 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 – Special 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 can get round this problem in one of two ways. The first is 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).

The alternative is to use triple-quoted strings, like this:

```
local var = '''dog's dinner''';
""""Right," says Fred. "That's quite enough of that, I think!"""";
```

Note that in the second example we have four double-quote marks in a row. Three of them mark the beginning and end of the string; the fourth is the double-quote mark we want displayed in the string.

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 formatter's 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, whereas 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."

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 mark-up that TADS 3 recognizes). 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 in <u>underlined</u>

- `<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 the 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.

If you are planning for your game to be played over the web, you should always use `aHref()` rather than the `<a> </a>` tags, since the latter will have their normal meaning when displayed in a browser (as a hyperlink to another web page, not a clickable shortcut to execute a command).

The above example introduces the final kind of special thing we can put inside 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 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. (assuming we had defined an `openDesc` method on Door):

```
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.

In addition to embedding expressions, the **<< >>** syntax can be used to vary text in other ways. We can, for example, vary the text displayed using an **<<if>> <<else if>> <<else>> <<end>>** construction such as:

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

We can also use various kind of **<<one of>> ... <<or>>** constructions to vary snippets of text randomly or cyclically, for example:

```
multiColouredCrystal: Thing 'multi-coloured crystal'
   "As you glance at the crystal it seems to sparkle <<one of>>red<<or>>blue
     <<or>>green<<or>>orange<<or>>purple<<at random>>. "
;
```

For a complete account of the **<< >>** constructs you can use in strings (and for further details of everything else we have covered in this section), see the String Literals chapter in the *TADS 3 System Manual*.

Although you can also use **<< >>** embedded expressions in single-quoted strings, there are one or two cases where they may not do exactly what you expect. For example, you should be aware of the difference between the property declaration:

```
ball: Thing 'ball'
    "It's round, as most balls are, and looks <<colour>>. "
    colour = '<<if isDirty>>a kind of muddy brown<<else>>bright red<<end>>'
    isDirty = true
;
```

And the similar-looking property-assignment statement:

```
    ball.colour = '<<if isDirty>>a kind of muddy brown<<else>>bright
        red<<end>>';
```

In the former case (the property declaration), the single-quoted string is evaluated every time the colour property is accessed, with the result that when **ball.isDirty**

changes from **true** to **nil**, the change will be dynamically reflected in the description of the colour of the ball. In the latter case (the assignment statement) the string expression will be evaluated only once, at the point when the assignment is made, with the result that a *string constant* will be stored in the property **ball.colour.** In this latter case, the description of the colour of the ball will not change when the value of **ball.isDirty** is changed.

Provided you keep this distinction in mind, using embedded **<< >>** expressions in single-quoted strings should be perfectly safe. In most cases, they *will* do what you want. More vexing cases can occur when embedded expressions are used in single-quoted strings that are elements of an EventList, but we'll worry about that when we come to EventLists.

4.5   TravelConnectors

We've already seen that a **Door** is a special kind of **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 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.moveInto(someDestination)**), travel round an adv3Lite 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 **conn.travelVia(gActor)** (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 classes from which `Room` inherits. A `Room` is a `TravelConnector` that always leads to itself.

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

- `destination` – the Room to which this TravelConnector leads.

- `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 *traveler* traveling via this connector; by default we simply display the travelDesc if the traveler is the player character.

- `travelDesc` – either a double-quoted string describing the travel or a method carrying out the side-effects of travel when the *traveler* is the player character; note that overriding `noteTraversal()` without calling `inherited` will disable this.

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

- `getDepartingDirection(traveler)` – returns the direction in which *traveler* would need to go to travel via this connector from the traveler's current location.

- `travelVia(traveler)` – causes *traveler* to travel via this connector. This can be useful to call on a Room, since in that case it also causes a display of the description of the room traveled to.

- `isConnectorListed` – (true or nil) determines whether this `TravelConnector` is to be listed in any list of exits (by default it is if it's visible).

- `isConnectorApparent` – (true or nil) determines whether this TravelConnector is apparent (i.e. not concealed and hence visible under proper lighting conditions).

- `IsConnectorVisible` – (true or nil) determines whether this TravelConnector is visible; by default it is if it's apparent and the lighting conditions are adequate (i.e. either the location or the destination is lit).

- **travelBarriers** – a single **TravelBarrier** object, or list of **TravelBarrier** objects, that applies to this **TravelConnector** (we'll say more about **TravelBarriers** below).

- **sayDeparting (traveler)** – display a message saying that traveler is departing via this connector.

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 a **Passage** object:

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

+ narrowTunnel: Passage '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! ";
   }

   travelDesc = "You just manage to squeeze through the narrow tunnel. "
;
```

Most of the time the direction properties of a room (north, east, south etc.) will point to TravelConnector objects of one sort or another (Rooms, Doors, Passages etc.), assuming they point to anything at all. But it's also legal for the direction property of a room to point to a double-quoted string, a single-quoted string or a method. If it points to a string, the string will be displayed (this can be used, for example, to explain why travel in that particular direction isn't possible). If it points to a method, the method will be executed (this could be used to carry out something other than standard travel, such as killing the player character if he sets off in a direction that leads him or her to fall down a mine shaft). If a direction property points to a method, that direction will be shown in the exit-lister, but if it points to a single-quoted or double-quoted string it won't be. If a direction property points to a method or double-quoted string then before travel notifications will be issued before the method is executed or the string displayed; these may result in the display or method being forestalled (we'll return to travel notifications in Chapter 13 below); but no travel notifications are issued when a direction property points to a single-quoted string. The three cases illustrated below are thus subtly different:

```
ledge: Room 'Ledge'
    "This narrow ledge runs round the side of the mountain. "
    east() { "You walk a short way to the east but you're forced to turn back
            when the path narrows to nothing. "; }
    north = "Better not; there's a sheer drop that way. "
    south = 'You can\'t walk through solid rock! '
;
```

In this example, east will show up as a possible direction of travel in the exit-lister, but north and south will not. Attempts to travel either east or north will result in before travel notifications being issued, but an attempt to travel south will not. This could make a difference if the player character were in the company of an NPC (non-player character) who might want to intervene in the player characters's travel attempts; the NPC would have the opportunity to do so in the event that the player character attempted futile travel to the east or perilous travel to the north, but not if the player typed SOUTH, a direction which the player character could not even attempt.

Using a method that simply displays something as the value of a direction property can thus be a useful way to add "soft boundaries" to the map, by making it appear that the game map is bigger than it is while explaining why the player character either cannot or does not want to travel in a direction that would in fact take him or her off your map.

Using a string or method to curtail travel is fairly straightforward; at first sight it may seem a lot more cumbersome to use a TravelConnector for such a purpose, since (on analogy with the various **Passage** objects), you might suppose we would have to do this kind of thing:

```
forestClearing: Room 'Forest Clearing'
    "A variety of paths runs of in all sorts of directions. "
    north = forestStreamConnector

;


forestStreamConnector: TravelConnector
   destination = byStream
   canTravelerPass(traveler)    {    return boots.isWornBy(traveler);   }

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

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 *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: Room 'Forest Clearing'
    "A variety of paths run off in all sorts of directions. "
    north: TravelConnector
    {
      destination = byStream
      canTravelerPass(traveler) { return boots.isWornBy(traveler); }
      explainTravelBarrier(traveler)
      {
        "That way is too muddy to walk down without a pair of sturdy boots. ";
      }
    }
;
```

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, connector) { return boots.isWornBy(traveler); }
    explainTravelBarrier(traveler, connector)
    {
      "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: Room 'Forest Clearing'
    "A variety of paths run off in all sorts of directions. "
    north: TravelConnector
    {
      destination = byStream
      travelBarriers = bootBarrier
    }
;
```

Another reason to define **TravelBarriers** might be if there were several different reasons why 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, connector) { return map.isIn(traveler); }
    explainTravelBarrier(traveler, connector)
    {
     "You'd better not go any further that way without a map. ";
    }
;

forestClearing: OutdoorRoom 'Forest Clearing'
    "A variety of paths run off in all sorts of directions. "
    north: TravelConnector
    {
      destination = byStream
      travelBarriers = [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.

One further point: in most cases the *traveler* parameter on the methods we've been considering will be the actor doing the travelling. But it's possible that the actor may be in or on a vehicle, in which case the *traveler* parameter will be the vehicle doing the travelling. To make something a vehicle we set its `isVehicle` property to `true`; we also need to make it possible for actors to get on or in it, which is something we'll come to later.

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 flashlight 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 not 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 an oak door leading south. 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). In the study is a pair of special shoes (make them of class Wearable).

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.

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. 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.

Across the river is a meadow, but you can only walk across the river if you're wearing the special shoes, and of course you can't ride the bike across the river.

A bicycle is parked in the drive. The player character can ride the bicycle on the level, but not up the tree or up or down stairs or around the boat. The player character can carry the bicycle but can't climb the tree while carrying the bicycle.

Even implementing a game as basic as this may require some features of TADS 3 and avd3Lite 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, with the aid of one further hint: this is how you might define the bicycle in outline:

```
bike: Platform 'bicycle;;bike'
   isVehicle = true
;
```

Filling in such details as the description is left as an exercise for the reader. When testing your game, note that there's no **ride** command defined in the library. To try riding the bike around, issue a **get on bike** command, and then use travel commands in the normal way (such as **north** or **go through door**). The player character should then travel around on the bike (until you issue a **get off bike** command).

If you like, you can compare your version with the Example 11.t file in the learning directory.

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'
   location = hall
;

redBall: Thing 'red ball' @hall
;

hall: Room 'Hall'
;

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

Things can also be picked up and moved to other rooms, or moved by authorial fiat using the `moveInto(newLoc)` method. But there's more to containment than this; both in the real world and in Interactive Fiction objects can 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 adv3Lite, to allow an object to have other things put in, on, under or behind it we can change its `contType` property to `In`, `On`, `Under` or `Behind`. More normally, though, we'll use a subclass of Thing that does that for us.

 In adv3Lite an object that can contain other objects will usually 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 an adv3Lite 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 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 some cases you might want to move objects around using the more elaborate `actionMoveInto(dest)` method. This would normally be the case where you're moving an object in direct response to a player's command and you need to carry out more checks on the possibility of the move. If, however, you want to move an object around by sheer authorial fiat, `moveInto(dest)` is good enough.

5.1.3 Defining the Initial Location of Objects

We can use one of three methods to define the initial location of objects that 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'
   location = hall
;

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

redPen: Thing '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' @hall;

yellowBox: Container 'small yellow box' @blueBox;

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

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

```
hall: Room 'Hall';

+ blueBox: Container 'large blue box';

++ yellowBox: Container 'small yellow box';

+++ redPen: Thing '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 more one plus sign than the object 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';

++ yellowBox: Container 'small yellow box';

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

++ greenBox: Container 'green box';

+++ blackPencil: Thing 'black pencil';

+++ whiteFeather: Thing 'white feather';

++ orangeBall: Thing 'orange ball';

+ oldHat: Wearable '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 place to another in our source code (using cut and paste), 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 six 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 directly or indirectly held by *actor*.

- `isDirectlyHeldBy(actor)` – determines whether the object is being directly held by *actor* (i.e. it's notionally in his/her hands rather than inside something else s/he may be carrying). 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.isDirectlyHeldBy(actor)` is `nil` (though `oldHat.isDirectlyWornBy(actor)` would then be true), but if the actor takes the hat off and continues to carry it `oldHat.isDirectlyHeldBy(actor)` becomes true. If the actor were to pick up the red pen directly (taking it out of the box) then `redPen.isDirectlyHeldBy(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.isDirectlyHeldBy(actor)` would be `nil,` while `redPen.isHeldBy(actor)` would remain true.

- `isDirectlyWornBy(actor)` – determines whether the object is being directly worn by the actor. Note that an object that's being worn is not considered as being carried, and *vice versa*.

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

```
    if(!orangeBall.isDirectlyHeldBy(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 four 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.

- **directlyHeld** – a list of objects directly held by the object.

- **directlyWorn** – a list of objects directly worn by the 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 returned 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';

++ yellowBox: Container 'small yellow box';

class Pen: Thing
   bulk = 2
;

+++ redPen: 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 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 bulk an actor can carry. To model this adv3Lite defines a `bulk` property and a `bulkCapacity` property on every `Thing`. 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 the total `bulkCapacity` 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` of an actor (or anything else) would be exceeded, since the default value is 10,000, but we can, of course, set it to something rather smaller if we wish.

By default the library defines the bulk of every `Thing` as 0, so if we want to track the total bulk being carried by actors or placed inside containers, it's up to us to give a meaningful bulk to (usually only portable) items according to whatever scale we deem appropriate, which depends on whether we want to regard the largest object in our game as being ten, a hundred or a thousand times bulkier than the smallest (say). If we also want to track the bulk of the player character and other actors (to limit what they can get inside) we'll probably need to use a scale with a larger range; a person is rather more than ten times bulkier than a pin.

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 character still has room in his or her notional hands, an object can't be inserted into a container (or picked up by a player) if its bulk exceeds the `maxSingleBulk` for that container or actor. By default `maxSingleBulk` is set to `bulkCapacity`, but it can be changed to something smaller if desired.

There's a further couple of points to note about bulk. To get the total bulk of the objects contained within something, we can call its `getBulkWithin()` method. To get the total bulk of all the items carried by an actor, however, we should use its `getCarriedBulk()` method. One difference is that anything being worn by an actor is not reckoned as being carried, so it doesn't contribute to the carried bulk. Another is that anything fixed in place is also not reckoned as being carried, so that too won't contribute to the total carried bulk; anything fixed in place in an actor's contents is likely to be a body part, not something carried.

5.1.7   Items Hidden in Containers

Sometimes an object inside a container may not be visible until we look inside the container (e.g. a small pin inside a large jar). One way we could represent this might be to define **isHidden = nil** on the pin and then make looking inside the jar call the pin's **discover()** method. But adv3Lite offers an easier way of dealing with this kind of situation by using the **hiddenIn** property. To hide one object in another, just list the hidden object in the **hiddenIn** list of the concealing object, while making the hidden object start out located nowhere (i.e. in nil, off-stage).

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'
  hiddenIn = [redPen]
;

redPen: Thing 'small red pen';
```

The containing object doesn't even have to be a container for this to work. For example, if we want to allow the player to find a gold coin hidden in a pile of junk we could just do this:

```
cellar: Room 'Cellar'
   "There's not much here but a pile of junk. "
;

+ junk: Fixture 'pile of junk[n]; rusty; bits metal; it them'
    "Rusty old bits of metal, mostly, but you never know what may lie concealed
    within. "
    cannotTakeMsg =' You certainly don\'t want to carry all that stuff around! '
    hiddenIn = [goldCoin]
;

goldCoin: Thing 'gold coin'
;
```

There is a slight difference in the way these two cases will behave, however. The command **look in yellow box** will result in the pen's being discovered and moved into the yellow box, while the command **look in junk**, or **search junk**, will result in the gold coin's being discovered and taken by the player character, since there's nowhere for it to go inside the junk. This behaviour can be controlled by two further properties, **findHiddenDest** and **autoTakeOnFindHidden**. If something isn't a Container but has a **hiddenIn** list, then the objects in the **hiddenIn** list are moved to **findHiddenDest** when they are discovered. By default, **findHiddenDest** is the actor doing the searching if **autoTakeOnFindHidden** is **true**, and the location of the concealing object otherwise, while **autoTakeOnFindHidden** is **true** if the concealing object is fixed in place and **nil** otherwise.

5.1.8 Notifications

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

Of these, `notifyRemove()` is the simpler, so we'll deal with it first. Whenever an object is about to be removed from inside another object , `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' @hall
    notifyRemove(obj)
    {
        "Removing <<obj.theName>> from <<obj.location.theName>>! ";
    }
;

+ greenBox: Container 'green box'
;

++ pebble: Thing 'pebble'
;
```

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:

```
modify Container
    notifyRemove(obj)
    {
        if(obj == pebble)
        {
            "The pebble refuses to leave <<obj.location.theName>>! ";
            exit;
        }
        else
            "Removing <<obj.theName>> from <<obj.location.theName>>! ";
    }
;

blackBox: Container 'black box' @hall
;

+ greenBox: Container 'green box'
;

++ pebble: Thing 'pebble;; stone'
;
```

This could result in the following transcript:

>take pebble

The pebble refuses to leave the green box!

>take green box

Removing the green box from the black box!

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.

The other method, `notifyInsert()`, works in much the same way. This can be illustrated via an extension to our previous example:

```
modify Container
    notifyInsert(obj)
    {
        "Putting <<obj.theName>> in <<theName>>. ";
    }

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



blackBox: Container 'black box' @hall

;

+ greenBox: Container 'green box'
;

++ pebble: Thing 'pebble;; stone'
;
```

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.

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 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 blob 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 the 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 wouldn't have to make this particular modification in practice since Thing already defines an **opened** property that does it for us, but it serves to illustrate the principle.

We sometimes need more control over where we inherit from. For example, suppose we want to define an object that behaves like a **Door** in just about every respect, except that we don't want opening one side to open the other side (perhaps because it's some special kind of double door). In that case we may want our custom door to used Thing's makeOpen method instead of Door's. We can do that by specifying which class we want to inherit from:

```
class CustomDoor: Door
   makeOpen(stat) {  inherited Thing(stat); }
;
```

Without that **Thing** following inherited we'd just inherit Door's **makeOpen()** method, which is precisely what we're trying to circumvent here.

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:

```
modify Topic
    owner = []
    nominalOwner() { return delegated Thing; }
    ownedBy(obj) { return delegated Thing(obj); }
;
```

The **Topic** class (which we'll encounter again later) inherits from the **Mentionable** class, but not from the **Thing** class, which defines **nominalOwner** and **ownedBy()**. So if (for whatever obscure reason) we wanted to **Topic** to be able to use the **nominalOwner()** and **ownedBy()** methods defined on Thing, we should need to borrow both these 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 **Container** class. 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:

- **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.

- **Booth** – a container that can also hold people, and that actors can get in and out of (we'll come back to this kind of container in Chapter 11).

- **OpenableContainer** – a container that can be opened or closed, and usually hides its contents when closed (but we can make it transparent if we wish by defining **isTransparent = true** on it). 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 **isOpen** property to true in the object definition.

- **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 thus little obstacle to a player opening a **LockableContainer**. The **isLocked** 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.

The use of these various classes shouldn't present any particular problems, but an example may be helpful here:

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

++ document: Thing '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 the 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   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 **Fixture** class. So, for example, we might have:

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

++ tray: Surface 'tray'
;

+++ mat: Surface 'mat'
;
```

```
++++ bowl: Container 'bowl'
;

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

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).

Just as there's a `Booth` class corresponding to the `Container` class, so there's a `Platform` class corresponding to the `Surface` class; a `Platform` is a `Surface` that actors can get on and off.

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 adv3Lite defines the following classes:

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

- `RearContainer` – something we can put things behind.

To hide things under or behind other things we can use the `hiddenUnder` and `hiddenBehind` properties in just the same way we use `hiddenIn`. Just as `hiddenIn` can be used with a `Container`, but needn't be, so `hiddenUnder` and `hiddenBehind` can be used with an `Underside` or `RearContainer`, but needn't be. This can be very handy when we want to hide things under or behind other things that we otherwise don't really want to make Undersides or `RearContainers`.

For example, we might have silver coin hidden under a rug. Ideally we'd like to make the rug a `Platform`, since it's clearly something the player could stand on, but it can't be both a `Platform` and an `Underside` at the same time. Fortunately, since we can use the `hiddenUnder` property, that's no problem:

```
+ rug: Platform 'rug; small dark'
    initSpecialDesc = "A small dark rug lies on the floor. "

    hiddenUnder = [silverCoin]
;

+ mirror: Thing 'mirror'
    initSpecialDesc = "A square mirror is hanging on the wall. "

     hiddenBehind = [bankNote]
;

silverCoin: Thing 'silver coin'
;

bankNote: Thing 'banknote; bank; note'
```

```
;
```

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 is likewise revealed.

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. By default **objInPrep** takes its value from the **prep** property of the Thing's **contType**, (e.g. 'in' for **In**, 'on' for **On**, 'under' for **Under**, and 'behind' for **Behind**). This property can be used to tweak certain kinds of message describing the whereabouts of an object. For example we could make things be listed as 'beneath' something rather than 'under' something, say, simply by changing **Underside.objInPrep** to 'beneath'.

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 adv3Lite world model seems not to allow for this. It turns out that adv3Lite 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 virtually 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 'wallpaper; faded pink'
    "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 'wallpaper; faded pink'
    "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 identifier 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: Room 'meadow'
    "The ground becomes distinctly marshier to the north. "
    north: TravelConnector
    {
        destination = marsh
        travelDesc = "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 `TravelConnector` 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 `TravelConnector` 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 `TravelConnector` 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 or between the class list and the opening brace (as is also the case when defining an ordinary object with the brace notation).

A nested object can be defined with properties and methods just like any ordinary

object, for example:

```
desk: Surface, Heavy 'desk;;furniture'
    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: Room 'meadow'
    "The ground becomes distinctly marshier to the north. "
    north: TravelConnector
    {
      destination = marsh
      travelDesc ="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   Multiple Containment

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 adv3Lite solution is the use of remapXX properties: **remapIn**, **remapOn**, **remapUnder** and **remapBehind**. Each of these can be used to refer to an object to which actions appropriate to a **Container**, **Surface**, **Underside** or **RearContainer** should be redirected or *remapped*.

This could be used to set up a desk with a drawer, for example:

```
desk: Heavy, Surface 'desk' @study
    "It has a single drawer. "
     remapIn = drawer
```

```
;

+ drawer: Fixture, OpenableContainer 'drawer'
   cannotTakeMsg = 'You can't take the drawer; it\'s part of the desk. '
;
```

With this set-up you can put things on the desk, but any attempt to put something in the desk, look in the desk, open the desk or close the desk will result in the corresponding action being remapped to the drawer.

This is all very well for the situation of the desk with a semi-separate drawer, but what happens if we want to be able to put things in, on, under and behind the same object, such as a low free-standing cabinet? In this case we can define some or all of the remapXXX properties as nested anonymous objects of the `SubComponent` class, like this:

```
+ cabinet: Heavy 'cabinet'
    remapOn: SubComponent { }
    remapIn: SubComponent
    {
        bulkCapacity = 10
        isOpenable = true
    }
    remapBehind: SubComponent { }
    remapUnder: SubComponent { 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 `SubComponents` representing the spaces in, on, under and behind the cabinet. Each of these `SubComponents` automatically takes its name from its `lexicalParent`, the cabinet, so that objects within these `SubComponents` are described as being in, on, under or behind the cabinet. None of these `SubComponents` has any `vocab`, so any commands targeted at the cabinet will be fielded by the `cabinet` object; but since the cabinet has a the appropriate `remapXXX` properties, it automatically redirects certain actions to the objects defined on its `remapXXX` properties, provided they're present. For example, **open, close, lock, unlock, put in** and **look in** are all directed to the `remapIn` object; **put on** is redirected to the `remapOn`; **put under** and **look under** are redirected to the `remapUnder`; and **look behind** and **put behind** to the `remapUnder` (provided objects have been defined on the relevant properties). Note also that we don't have to specify that these four objects are respectively a `Surface`, `Container`, `Underside` and `RearContainer`; the library can work that out for itself from the properties to which our nested anonymous objects are attached.

We may often want some objects to start out in one or other part of a multiply-containing object of this sort. 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'
    location = cabinet.remapOn
;

paper: Thing 'piece of paper'
   location = cabinet.remapBehind
;

mat: Surface 'mat'
   location = cabinet.remapIn
;

coin: Thing 'coin'
    location = cabinet.remapUnder
;
```

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 (as we have seen before) 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: Heavy 'cabinet'
    remapOn: SubComponent { }
    remapIn: SubComponent
    {
        bulkCapacity = 10
        isOpenable = true
    }
    remapBehind: SubComponent { }
    remapUnder: SubComponent { bulkCapacity = 10 }
;

++ vase: Container 'vase'
    subLocation = &remapOn
;

++ paper: Thing 'piece of paper'
    subLocation = &remapBehind
;

++ mat: Surface 'mat'
    subLocation = &remapIn
;

++ coin: Thing 'coin'
    subLocation = &remapUnder
;
```

While this doesn't save a huge amount of typing, it does make it easier to associate

objects contained in a multiply-containing object with that object in the way we lay out the code.

Note that we don't need to define all four `remapXXX` properties on any given object; 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: Heavy 'table; kitchen'
    remapOn: SubComponent { }
    remapUnder: SubComponent { }
;

+ washingMachine: Heavy 'washing machine'
    remapOn: SubComponent { }
    remapIn: SubComponent
    {
        notifyInsert(obj)
        {
            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 multiple containment 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 'briefcase; light brown'
    "It's a light brown case with a handle and combination lock. "
;

+ Fixture 'handle' // DON'T DO THIS!
;

+ Fixture 'lock; combination' // 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 use `remapIn` to represent the inside of the briefcase; we should start out with something like this:

```
briefcase: Thing 'briefcase; light brown'
    "It's a light brown case with a handle and combination lock. "
    remapIn: SubComponent, LockableContainer { }
;

+ Fixture 'handle'
;

+ Fixture 'lock; combination'
;
```

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

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. 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 adv3Lite, we've now covered the fundamentals of the adv3Lite 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 adv3Lite (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 involves 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 adv3Lite allow 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 adv3Lite 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 adv3Lite:

- `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.

- `gCommand` – the current command. We'll explain the notion of a command below.

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 Something.

- `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 adv3Lite library will then appear in the bottom left-hand panel. Alternatively, you can look up the action reference section in the *adv3Lite Manual*.

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 `TravelAction`, not a `TIAction`. In adv3Lite *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 adv3Lite 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 a 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 as `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.curDobj`, 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.curDobj)
```

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.curDobj)`. Note that this replacement is cumulative; having replaced `gDobj` with `(gAction.curDobj)` the preprocessor will replace `gAction` with `(libGlobal.curAction)` so that the full expansion of `gDobj` becomes `((libGlobal.curAction).curDobj)`; thus whenever we write `gDobj` in our source code, `((libGlobal.curAction).curDobj)` 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 adv3Lite game source files:

```
#include "advlite.h"
```

This ensures that we can use all the macros defined in the adv3Lite 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 been 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 arguments, 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 goInstead(dirn)   doInstead(Go, dirn##Dir)
```

Then when the processor comes across `goInstead(north)` it will replace it with `doInstead(Go, northDir)`.

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
```

For example, consider the following code (of a kind that's very common when we start customizing and defining actions):

```
dobjFor(Take)
{
    preCond = [touchObj]
    verify()
    {
        if(isIn(gActor))
          illogicalNow('You are already holding it! ');
```

```
    }
    check() { }
    action()
    {
        actionMoveInto(gActor);
    }

    report()
    {
        "Taken. ";
    }
}
```

This is exactly equivalent to:

```
    preCondDobjTake = [touchObj]

    verifyDobjTake()
    {
        if(isIn(gActor))
          illogicalNow('You are already holding it! ');
    }

    checkDobjTake() { }

    actionDobjTake()
    {
        actionMoveInto(gActor);
    }

    reportDobjTake()
    {
        "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 Jump
    execAction(cmd)
    {
        if(gActor.bulk > 1500)
            "You're too big 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 `Sleep` as:

```
DefineIAction(Sleep)
    execAction(cmd)
    {
        DMsg(no sleeping, 'This {dummy} {is} no time for sleeping. ');
    }
;
```

So that if we want to change the way the Sleep action works, we might do better to change the 'no sleeping' message (how to do this is a subject to which we shall return). 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.

Incidentally, the _cmd_ parameter to the `execAction(cmd)` method refers to the current `Command` object, which contains information about how the parser interpreted the command the player typed. We'll say more about Command objects later.

6.3.2   Actions With Objects

If an action has a direct object, or both a direct object and an indirect object, then we **_don't_** override `execAction(cmd)`; instead 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 called directly or indirectly from the action's `execAction()` method, which is why we shouldn't override it); 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 non-portable 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 the pseudo-action, `Default`. If we define `dobjFor(Default)` or `iobjFor(Default)` verify handlers on a class or action, these handlers will be used on objects for which `isDecoration` is true (normally the `Decoration` class and its subclasses) where the action was not listed in the object's `decorationActions` property. The default definition on `Thing` of `decorationActions` is `[Examine, GoTo]`, which means that the commands EXAMINE X and GO TO X work normally when X is a Decoration, but that any other command directed to X will be met with the response "The X is not important".

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; ripe; food fruit'
    dobjFor(Eat)
    {
        remap = xxx
        preCond = [ ... ]
        verify() { ... }
        check() { ... }
        action() { ... }
        report() { ... }
    }
;
```

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, action and report stages in turn. In fact, many 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. Also, if the action involves several objects (e.g. **take bell, book and candle**) then the `report()` stage is only executed once at the end, so that it can report on the complete set of actions (e.g. "You take the bell, the book and the candle.")

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 the same action on a different object. If you want this kind of remapping, you simply set the remap property to the other object you want the action diverted to.

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'
    "The desk has a single drawer. "
    dobjFor(Open) { remap = drawer }
;

+ drawer: OpenableContainer, Fixture 'drawer'
;
```

Of course we'd probably use `remapIn = drawer` here to divert a whole set of actions at once, but the example serves to illustrate the principle. In fact the library uses just this mechanism to make `remapIn` work, for example:

```
dobjFor(LookIn)
 {
    remap = remapIn
    ...
 }
```

If `remapIn` is `nil` then `remap` is nil, so no remapping takes place; but if `remapIn` points to another object, then so does `remap`, and the remapping goes ahead for LookIn and all the other relevant actions.

Note that this is the *only* kind of remapping that remap can do in adv3Lite. If you want to change one action into a totally different action, then the best way to do it is with a `Doer`, which is something we'll come to later.

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 decides 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'
    dobjFor(Eat)
    {
        verify() {}
    }
;

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

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

```
banana: 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: Actor 'Mary;; woman; her'

    dobjFor(Kiss)
    {
        verify() {  logicalRank(120); }
    }
;
```

With this definition, **kiss woman** will be taken to mean **kiss mary** even if other women are present, although an explicit **kiss anne** command will still be allowed (provided Anne is present). As is apparent from this example, the `logicalRank()` macro takes one arguments: 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.

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).

- `implausible(msg)` – disallow this action with this object. This is almost the same as `illogical` except that it's regarded as slightly less illogical, so that the parser will prefer an implausible choice to an illogical one.

- `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 a check() routine to block an action all it needs to do is to display a message explaining why the action is being disallowed.

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'
    wornBy = me
    dobjFor(Doff)
    {
        check()
        {
            "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'
    wornBy = me
    dobjFor(Doff)
    {
        check()
        {
            if(!me.isIn(myBedroom))
                "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 possibly 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'
   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'
   dobjFor(Push)
   {
      action()
      {
          slidingDoor.makeOpen(!slidingDoor.isOpen);
          "You push the green button and the door slides
           <<slidingDoor.isOpen ? 'open' : 'closed'>>. ";
      }
   }
;
```

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).

Another approach when we want the outcome of an action to depend on a particular pairing of objects is to use a **Doer**, which we'll discuss in due course.

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 'antique vase; delicate glass'
    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.

At this point we should pause to consider the effect of displaying something at the action stage. Normally the library's standard report for an action (such as a laconic 'Dropped') is produced at the **report()** stage (which we'll come to next). Displaying something at the action stage suppresses what would have been displayed at the report stage for that object (so, in the example above, we won't get the 'dropped' message as well). If we want something displayed at the action stage to be in addition to what the report stage is going to report, then we need to use either the **reportAfter()** macro or the **extraReport()** macro, like this:

```
vase 'antique vase; delicate glass'
    dobjFor(Drop)
    {
        action()
        {
            inherited;
            extraReport('(Remembering to be very careful with the vase)');
            reportAfter('You feel most relieved to have set the vase
          down without breaking it. ')
        }
```

```
    }
;
```

This might produce an output like this:

>drop vase
(Remembering to be very careful with the vase)
Dropped.
You feel most relieved to have set the vase down without breaking it.

Or possibly this:

>drop all
(Remembering to be very careful with the vase)
You drop the penknife, the vase, the big red ball and the walking stick.
You feel most relieved to have set the vase down without breaking it.

Two methods it's useful to know about in relation to action routines are **doNested()** and **doInstead()**. Both of these can be used to carry out some other action; the difference is that the action routine will continue after **doNested()** but not after **doInstead()** (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: Fixture 'sliding door'

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

Note that **doInstead()** and **doNested()** are both methods, not functions or macros, so they can only be used on objects that define them. They are in fact defined on the **Redirector** class, from which **Thing**, **Doer** and **Action** inherit. They can thus only be used from methods of these three classes, or of classes or objects that descend from any of these three classes.

6.3.8   Report

The sole function of the report stage is to report on the action that's just taken place. The report produced should be a default kind of report – the standard report for that kind of action – and it should not only be applicable to any object that might be involved in the action, but to all of them (in a case where the same action, such as **take all**, may be iterated over a number of objects).

A report routine should thus never be written to report on a specific object. If you want a customized report for a specific object this should be defined in the action() routine of the object concerned. It follows that it never really makes sense to define a report routine anywhere other than on the Thing class. It certainly makes little or no sense to define it on a particular object, since in the general case you can never be sure which object's **report()** routine will be called (one possible exception might be when the grammar of the action only allows it to act on a single object at a time). The function of a report routine is to give a summary report of the effect of the action on all the objects the action iterated over, e.g. "You take the ball, the bat and the gloves"), and to do this it will be called only on the last object in such a list; in general you won't know what that last object will be. This is why report routines should generally only be defined on the Thing class, so that the same version will be called no matter what Thing it happens to be called on.

It follows that you need some way of referring to all the objects an action iterated over. The library defines special macro for this purpose, called **gActionListStr**. Once an action reaches the report stage (but not before) this macro will evaluate to a single-quoted string containing a formatted list of the objects involved in the command, e.g. 'the ball, the bat and the gloves'. It can thus be used like this:

```
modify Thing
   dobjFor(Take)
   {
      report()
      {
           "You carefully pick up <<gActionListStr>>. ";
      }
   }
;
```

Any report routine you write should basically follow this pattern (though there are slightly more sophisticated ways of doing it we'll encounter in due course). In particular a report routine should never (normally) be written with a string like **"You carefully pick up <<gDobj.theName>>"**, since this will only refer to what happens to have been the last object in the list, not the whole list.

It's also possible to make the list of objects the subject of a sentence like this, but that requires techniques we haven't covered yet.

6.3.9 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 adv3Lite 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.

- **objAudible** – the object must be audible 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).

- **containerOpen** – the object must be open for the action to proceed, but only if it's a container (i.e. if its **contType** is **In**); an implicit open 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.

- **objDetached** - the object must be unattached to any other object for the action to proceed (an implicit detach action is attempted if not).

- **objNotWorn** – the object must be unworn for the action to proceed (an implicit doff action is attempted if not).

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: 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 want to know more, you could read the section on Actions in the *adv3Lite Manual*.

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:

```
modify Jump
   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.moveInto(spareBed);
             gActor.getOutermostRoom.lookAroundWithin();
        }
        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.moveInto(spareBed);
             gActor.getOutermostRoom.lookAroundWithin();
             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";
```

The for loop can take also take the form of `for..in` or `for..in range`. For example, the loop in the previous example could be written:

```
for(local i in 1..10)
    "<<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   Commands and Doers

6.5.1   Commands

When the player enters a command at the command prompt, the parser creates one or more `Command` objects to represent its interpretation of what the player meant. Each Command object encapsulates information about what action the parser interprets the player as wanting to carry out, what objects are involved in that action, and certain other information about what the player typed. The Command object then passes this information to a Doer, which can either execute the command unaltered, or transform it into a completely different command, or simply stop it in its tracks. The execution of the command may then involve the execution of one or more actions. For example the command **look in box** may involve an implicit Open action followed by the explicitly requested LookIn action.

The current `Command` object is passed as the `curCmd` parameter to the Doer's `execAction(curCmd)` method and thence as the `cmd` parameter to the action's `execAction(cmd)` method. You can also get at the current command object with the `gCommand` pseudo-global variable.

You can probably spend quite some time writing a game in adv3Lite without having to worry about `Command` objects, but there can be times when it's useful to get at the information contained in some of their more straightforward properties. These include:

- `actor` – the actor performing the command
- `action` – the action the command thinks should be carried out
- `dobj` – the direct object of the action the command thinks should be carried out
- `iobj` – the indirect object of the action the command thinks should be carried out
- `dobjs` – a Vector containing the `NPMatch` objects relevant to all the direct objects to be interated over.
- `verbProd` – the VerbProduction (VerbRule) matched by this command.

Most of these require a bit of further explanation.

First of all, the command's `action`, `dobj` and `iobj` properties may or may not correspond to `gAction`, `gDobj` and `gIobj`. It all depends on the type of action and what's intervened. For example, if an Open action triggers an implicit Unlock action, then `gAction` may be `Unlock` while `gCommand.action` may be `Open`. Again, the command object's interpretation of `dobj` and `iobj` is unlikely to be that of the action's when the action is a TopicAction, TopicTAction, LiteralAction or LiteralTAction. In

particular, while `gDobj` and/or `gIobj` may be `nil` for these kinds of action, `gCommand.dobj` or `gCommand.iobj` will contain the `LiteralObject` or `ResolvedTopic` that the command matched, in which case the name property of the `LiteralObject` will contain a form of the literal text entered by the player, while the `topicList` property of the `ResolvedTopic` will contain a list of Topics that the player's command might match.

As noted above, the `dobjs` property contains a Vector of NPMatch objects. The actual objects concerned will be contained in the `obj` property of the NPMatch object. This means that we can get at a list of objects to be iterated over by looking at `gCommand.dobjs[1].obj`, `gCommand.dobjs[2].obj` and so forth, or we can iterate over them with code like:

```
foreach(local cur in gCommand.dobjs)
{
    local o = cur.obj;
   /* do something with o */
}
```

The `verbProd` property contains an object that has a number of useful properties. Its `grammarTag` property identifies the VerbRule that was matched (for VerbRules, see later in this chapter). Its `tokenList` property contains a list of the tokens that make up the command typed by the player. And for a command that involve a direction (e.g. **go east**, **push box west**, **throw ball south**) the `dirMatch.dir` property (i.e. `gCommand.verbProd.dirMatch.dir`) gives the direction that was matched (e.g. `eastDir`, `westDir` or `southDir`).

Some awareness of these properties of the `Command` object can be particular important when we come to define a `Doer`, since the Command object is passed to a Doer's `execAction(curCmd)` method via the `curCmd` parameter, and the method will often need to analyse that command to decide what to do with it.

6.5.2   Doers

A Doer is an object that stands between a Command and an Action and decides how the Command is to be translated into one or more Actions. The library defines four default Doers that simply pass the command straight through to the associated action unaltered, but game code can include custom Doers to change what a command does. (If you're familiar with Inform 7, think of a Doer as adv3Lite's rough equivalent to an instead rule).

A Doer can make a command carry out a completely different action from normal, or it can stop an action in its tracks. We can specify precisely what commands we want a particular Doer to match, and under what circumstances. Where more than one Doer could potentially match the same command, the most specific Doer is the one that's

used.

We make a Doer match a command by defining its `cmd` property. This is a single-quoted string that contains more or less what the player would type to trigger the command, except that we replace the vocab of any objects involved with their programmatic names (e.g. `redBall` or `table`). We can also use the name of a class instead of the name of an object. The following are all valid cmd strings to use with a Doer:

```
'jump'
'put redBall in greenBox'
'put Thing in Container'
'take stick|hat'
```

The last of these would match attempts to take either the stick or the hat. If we want to match more than one verb, however, we have to list the commands separated by semicolons:

```
'put redBall in greenBox; eat redBall'
```

Since every Doer we'll ever define must define a `cmd` property, we can do it via a template, thus:

```
Doer 'put redBall in greenBox'
;
```

Of course none of this is very useful unless we make the Doer actually do something. To do this we generally override its `execAction()` method, either to make it execute a different action, or to stop the action altogether (we could also write our own action-handling from scratch in this method, but unless we want to do something very unusual, that's probably more trouble than it's worth).

For example, suppose we have a door that can be unlocked by inserting a card into a slot. In this case we might want **put card in slot** to work the same as **unlock door with card**. We can do this by calling the `doInstead()` method from within the `execAction()` method, like so:

```
Doer 'put card in slot'
   execAction(c)
   {
       doInstead(UnlockWith, securityDoor, card);
    }
;
```

The `doInstead()` method can take one, two or three arguments. The first argument is the action you want to perform. The second and third arguments are the direct and indirect objects you want to perform the action on, provided it's the kind of action that takes these objects.

To stop an action in its tracks we can just display a message followed by an `abort`

statement, like so:

```
Doer 'jump'
   execAction(c)
   {
       "Jumping is such an unseemly waste of energy! ";
       abort;
   }
;
```

Actions involving motion in particular compass directions are a special case. We can match them by using a cmd string consisting of the word 'go' followed by one or more direction names, separated by vertical bars, like so:

```
Doer 'go east|west|north|south'
   execAction(c)
   {
       "You're a bishop, and bishops can only move diagonally. ";
       abort;
   }
;
```

In this case the direction names have to exactly match the way they would appear in the exit lister.

To synthesize a command to travel in a particular direction we can either use `doInstead()` with `Go` and the name of the direction object (e.g. `northDir`) or just `goInstead()` with the name of the direction. The two are exactly equivalent:

```
Doer 'jump'
   execAction(c)
   {
       doInstead(Go, upDir);
   }
;
```

Or alternatively,

```
Doer 'jump'
   execAction(c)
   {
       goInstead(up);
   }
;
```

All the Doers we've seen so far are active all the time, but we can also control when they take effect by defining one or more of the following properties on a Doer:

- `when` – a condition that must be true for the Doer to take effect.

- `where` – a Room or Region or a list of Rooms and/or Regions one of which the actor must be in for the Doer to take effect.

- `who` – an actor or a list of actors, one of which must be the current actor for the Doer to take effect.

- **during** – a Scene, or a list of Scenes, one of which must be currently in progress for the Doer to take effect. (We'll discuss Scenes fully in a later chapter).

For example, to prevent the player character from jumping while in the **caveRegion** (and here at last is one use for Regions) we could do this:

```
Doer 'jump'
   execAction(c)
   {
       "There's not enough headroom to jump here. ";
        abort;
   }

   where = caveRegion
;
```

Or to stop the player from issuing an **undo** command during a fight scene we could do this:

```
Doer 'undo'
   execAction(c)
   {
       "Sorry; UNDO is disabled during this fight scene. ";
        abort;
   }

   during = fightScene
;
```

Where several Doers could match any given action, the most specific one wins. A Doer is more specific the more conditions (when, where, who, during) it imposes and the more specific the objects it matches (individual objects are more specific than classes, and subclasses are more specific than parent classes). If we need to, we can override this ordering by setting the Doer's **priority** property. The Doer with the highest priority always takes precedence; the default value of the **priority** property is 100.

6.6   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(Ponder)
    execAction(cmd)
    {
        "You ponder deeply, 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. Remember, though, that you should never override the `execAction()` method on actions that do take objects, i.e. TActions, TIActions, TopicTActions and LiteralTActions.

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
    : VerbProduction
    action = Cross
    verbPhrase = 'cross/crossing (what)'
    missingQ = 'what do you want to cross'
;
```

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 `VerbProduction`, is necessary for every VerbRule, defining it to belong to the VerbProduction class. Next we define the action with which this `VerbRule` is associated by assigning it to the `action` property. 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.

Following the class name, we can define other properties and methods in the normal way, but the only other properties we generally need to define here are the `verbPhrase` and the `missingQ`. The library uses the `verbPhrase` to construct message relating to the action such as "(first trying to cross the river)" 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 'what' we want used in posing questions about the action ("Whom do you want to ask?" or "What do you want to cross?"). Finally the `missingQ` defines the question the parser can ask if the player types **cross** without specifying an object 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
   : VerbProduction
    action = Cross
    verbPhrase = 'cross/crossing (what)'
    missingQ = 'what do you want to cross'
;
```

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

```
VerbRule(Think)
   'think' | 'ponder' | 'cogitate'
   : VerbProduction
   action = Think
   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
   : VerbProduction
   action = OpenWith
   verbPhrase = 'open/opening (what) (with what)'
   missingQ = 'what do you want to open; what do you want to open it with'
   dobjReply = singleNoun
```

```
    iobjReply = withSingleNoun
;
```

In this case note that the `missingQ` comprises two sections, divided by a semicolon; the first to ask for a missing direct object and the second to ask for a missing indirect object.

Using `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(cmd)` 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 subj dobj} {is} not something {i} {can} cross. '

    dobjFor(OpenWith)
    {
        preCond = [touchObj]
        verify() { illogical(cannotOpenMsg); }
    }

    iobjFor(OpenWith)
    {
        preCond = [objHeld]
        verify() { illogical(cannotOpenWithMsg); }
    }
    cannotOpenWithMsg = '{I} {cannot} open anything with {that iobj}. '
;
```

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 subj dobj} {is} not something {i} {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'
    cannotCrossMsg = '{I} {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 used `cannotOpenMsg` without actually defining it; that's because there's already a `cannotOpenMsg` defined on Thing in the library, and we can make use of it here.

Another point to note is that our messages contain lots of strange looking pieces of text in curly braces, like `{I}` or `{That subj dobj}`. These are *message parameter strings*. When the text is actually displayed the library substitutes text appropriate to the circumstance. For example `{I}` 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 subj dobj}` 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/d}` or `{es/ed}` and the like at the end of other verbs to secure similar agreement, or, in the case of irregular verbs, enclose the complete verb in curly brackets e.g. `{I} {put} down {the dobj}.` ' Using these message parameter strings helps makes our responses as general as possible. Other commonly useful ones include:

- `{the subj dobj}` – the name of the direct object preceded by the definite article ('the'), as the subject of the sentence.

- `{a subj dobj}` – the name of the direct object preceded by the indefinite article ('a' or 'an'), as the subject of the sentence.

- `{the dobj}` – the name of the direct object preceded by the definite article ('the'), as the object of the sentence.

- `{he dobj}` – the correct pronoun for the direct object in the subjective case ('he', 'she', 'it' or 'they')

- `{him dobj}` – 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'
    globalParamName = 'banana'
;
```

We can then use parameters substitution strings like `{the subj banana}` or `{him`

banana}. 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 subj obj} {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. It follows that when a parameter substitution occurs at the start of a sentence, you should start it with a capital letter to make sure the sentence starts with a capital letter. For the full story on message parameters, including a list of all the ones the library defines, see the chapter on 'Messages' in the *adv3Lite Manual*.

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()
        {
            "{I} {set} out across {the dobj}. ";
             connector.travelVia(gActor);
        }
    }
;
```

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 section on Actions in *The adv3Lite 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!

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.

An adv3Lite 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: Actor 'Bob;; man; him'
    seenProp = &bobHasSeen
;
```

```
carol: Actor 'Carol;; woman; her'
    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 the game 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 is what 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 `familiar` property, analogous to the `seen` property which we have just met. We can set the `familiar` 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, `familiar` is `nil` on everything, but if the player character starts out the game knowing about several things, we can define `familiar` as `true` on those objects.

So why do we call this property `familiar` rather than `known`? The reason is that the player characters is reckoned to know about something either if it is `familiar` or if has been `seen`. There is a `known` property, but it's true if either `familiar` is `true` or `seen` is `true`.

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: Actor 'Bob;; man; him'
    seenProp = &bobHasSeen
    knownProp = &bobKnows
```

```
;

carol: Actor 'Carol;; woman; her'
    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 **familiar** 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 an adv3Lite 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'
  dobjFor(Open)
  {
    check()
    {
      if(isStuck)
        "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 you're entirely free to use this mechanism for any purpose you find helpful.

By default, the library simply records that fact *that* a particular tag has been revealed. We can, if we like, associate more information about the revealing of tags by overriding the **setRevealed(tag)** method on **libGlobal**, perhaps to store the turn on which the revelation took place, or the location, or some more complex set of data encapsulated in an object of our own devising.

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 Code 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, ... item*n*]**

- 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 Enumerators 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: Heavy, OpenableContainer
;
```

Then **box.getSuperclassList()** would return **[Heavy, OpenableContainer]**, while **box.ofKind(Heavy)**, **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(Actor)** 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.
   }
;
```

7.2.6   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*.

7.3 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 `vocab` (which works in precisely the same way as the `vocab` property on Thing). So we might, for example, define:

```
tWeather: Topic vocab= 'weather';
tChinesePolitics: Topic vocab = 'chinese politics';
tMeaningOfLife: Topic vocab = 'meaning of life';
```

Since we always need to define the `vocab` 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 of life';
tMagicFormula: Topic 'magic formula' @nil;
```

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 our 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 `familiar`, which has the same meaning as the `familiar` property on `Thing`. Whereas `Thing.familiar` is nil by default, `Topic.familiar` starts out as true, so that if there are topics the player character starts the game not knowing about, we need to change `familiar` to nil on

those topics; we can do this via the template as in the `tMagicFormula` example above. 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 `topicDobj` 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 `topicDobj` or `topicIobj` token in its `VerbRule` (a `TopicAction` or `TopicTAction`), the object matching the `topicDobj/topicObj` 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()` or just `gTopicMatch`. This is only *probably* the simulation object or `Topic` in question, since a `ResolvedTopic` actually maintains a list of possible matches (in its `topicList` property) and `getBestMatch()` somewhat arbitrarily returns the first item from this list. 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 `topicXobj` token, but `gTopic.getBestMatch()` will return a Topic newly created to match the player's input.

7.4 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
   : VerbProduction
   action = Pick
   verbPhrase = 'pick/picking (what)'
   missingQ = 'what do you want to pick'
;

modify Thing
  dobjFor(Pick)
  {
     preCond = [touchObj]
     verify() { illogical(cannotPickMsg); }
  }
  cannotPickMsg = '{That dobj} {is} not something {i} {can} pick. '
;

class Apple: Food 'apple'
;

orchard: Room 'Orchard'
  "An apple tree grows in the middle of the orchard. "
;

+ tree: Fixture 'apple tree'
;

++ Fixture 'apple'
   dobjFor(Pick)
   {
      verify() { }
      action()
      {
         local apple = new Apple;
         apple.moveInto(gActor);
         "You pick an apple from the tree. ";
      }
   }
;
```

Here the **Fixture** 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 (not because they're dynamically created, but because you end up with so many objects in scope at once).

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;
        vocab = name + ';;fruit ';
        inherited();
    }
    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*.

7.5 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 adv3Lite 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:

```
+ Consultable '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 **look up 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 something 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 adv3Lite, 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. If 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 '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, &senseProp?, source?);
```

The two extra (but optional) parameters are *senseProp* and *source*. 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 *senseProp,* which must be one of `&canSee` (sight), `&canHear` (sound), `&canSmell` (smell) or `&canReach` (touch) (these are in fact methods of the Q object, which is used to query the world model about such things as scope and sensory connections, but we needn't go into that right now). The question-mark indicates an optional parameter: if we don't specify the *source* it's taken to be the same as *obj*, and if we don't specify the *senseProp* it's taken to be `&canSee`. We can specify *senseProp* without specifying *source*, but we can't specify *source* without also specifying *senseProp*. We could thus revise our dynamite accordingly:

```
dynamite: Thing 'stick of dynamite[n]'
    dobjFor(Burn)
    {
        verify() {}
        action()
        {
            "You set the dynamite alight. ";
            new SenseFuse(self, &explode, 3, &canHear);
        }
    }
    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, &senseProp?, source?);
```

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, &canHear);
```

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 Event 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. The keyword *new* is optional when creating an anonymous function. The following is equally legal:

```
local f = function(x, y) { return x + y; };
local sum = f(1, 2);
```

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 is 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.

It's also possible to create anonymous methods (and floating methods); we'll return to that briefly later on.

For a fuller account of anonymous functions (and methods), 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 passed.

- **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 (perhaps 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.moveInto(cellar);
            cellar.lookAroundWithin();
        }
    ]
;
```

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.moveInto(cellar);
        cellar.lookAroundWithin();
    }
;
```

Embedded expressions can be used in single-quoted strings. Why, then, should we ever want to use an anonymous function to encapsulate a double-quoted string in an Event List just so we can use an embedded expression? Surely we could just write something like:

```
myList: EventList
    [
        'A single-quoted string. ',
        'A double-quoted string with an <<embeddedExpression()>>. '
    ]
    embeddedExpression() { 'embedded expression'; }
;
```

Often, this will indeed work perfectly well. For example, there would be nothing at all wrong with an EventList constructed like this:

```
myList: ShuffledEventList
    [
        'The wind whistles in the trees. ',
        'A <<if frontDoor.isOpen>>loud<<else>>muffled<<end>>sound wafts
          through the <<frontDoor.name>>. ',
        'From <<me.location.theName>> you notice the wind change direction. ',
        'The sun glints off the <<if window.isBroken>>broken<<else>>front<<end>>
         window. '
    ]
;
```

And this type of thing probably covers the most common cases where you might want to use embedded expressions in an EventList. Note, however, that the following would not work at all how you might expect:

```
myList: ShuffledEventList
    [
        'The wind whistles in the trees; you have now noticed this <<++count>>
          times. ',
        'A <<one of>>starling<<or>>crow<<or>>pigeon<<or>>blackbird<<shuffled>>
          suddenly takes flight. ',
        'You notice the wind change direction to the <<one of>>north<<or>>
          south<<or>>east<<or>>west<<cycling>>. ',
        'The sun <<one of>>suddenly<<or>>once again<<stopping>> glints off
             the front window. '
    ]

    count = 0
;
```

If you wrote something like that, you'd find the `count` variable increasing far more rapidly that expected, the cycling of directions not working properly, the stopping list

reaching 'once again' prematurely, and the shuffled list of birds apparent not being shuffled properly. The (albeit somewhat obscure) reason for this is that the elements of an EventList are evaluated far more frequently than you might expect (typically six times per turn), even though you only see one of them displayed. In this kind of case, you do need to enclose the embedded expressions in double-quoted strings using anonymous functions. The following would work fine:

```
myList: ShuffledEventList
   [
       {: "The wind whistles in the trees; you have now noticed this <<++count>>
         times. "},
       {: "A <<one of>>starling<<or>>crow<<or>>pigeon<<or>>blackbird<<shuffled>>
         suddenly takes flight. "},
       {: "You notice the wind change direction to the <<one of>>north<<or>>
         south<<or>>east<<or>>west<<cycling>>. "},
       {: "The sun <<one of>>suddenly<<or>>once again<<stopping>> glints off
            the front window. "}
   ]

   count = 0
;
```

The rule of thumb is that it's safe to use an embedded expression in a single-quoted string in an EventList when the embedded expression won't change its value in the course of a single turn (even if it is evaluated several times during the course of that turn) but not otherwise. A `<<one of>>...` type construct will nearly always change its value with successive evaluations, and so should not be put inside a single-quoted string used as a list element. The same applies to other expressions that explicitly change their value each time they're evaluated (such as `++count`) or have other side-effects that might make successive changes to the game state over the course of a single turn.

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 `roomDaemon` method. If the Room is mixed in with an EventList class the default behaviour of roomDaemon is to call doScript, thereby driving the EventList, which will then automatically shows its elements every turn the player character is in the room in question. This could be used to display a series of atmospheric messages, for example:

```
class ForestRoom: Room, 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` might be useful is in conjunction with a `TravelConnector`. If the `TravelConnector` is also an `EventList`, then traversing it will automatically call its `doScript()` method (provided we haven't otherwise overridden its `travelDesc()` method). For example:

```
clearing: Room 'Forest Clearing'
    "It looks like you could go east or west from here. "
    west = streamBank
    east: TravelConnector, 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 *Adv3Lite 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 as 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 also 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 data types 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). Note that if we like we can use **for** in place of **foreach** in this kind of 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*.

Two further function to be aware of are **nilToList()** and **valToList**(); if the argument to either of these functions is a list, the function returns the list unchanged, but if the argument is nil the function returns the empty list **[]**. The **valToList**() function additionally turns just about anything fed to it into a list if it isn't a list already, so that (for example) **valToList(redBall)** returns the list **[redBall]**. This can be useful when we want to perform a list operation on a property that may contain either a list or a single object 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 = valToList(inherited) + objHeld
```

Apart from the `nilToList()` and `valToList()` functions, 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 it's 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 'bomb; long black; cylinder'
    "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()
```

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*. It is worth being aware of one potential trap with `static`, however, and that is that it should generally only be used to initialize individual object properties, not class properties. For example, were you to do this:

```
class Friend: Actor
    friends = static new Vector(10)
;
```

And then create a whole series of Friend objects, you'd find that they all shared the same `friends` Vector, which probably wasn't at all what you intended. What you'd probably need in a case like this is something like:

```
class Friend: Actor
    friends = perInstance(new Vector(10))
;
```

This would ensure that every instance of the Friend class ended up with its own friends Vector.

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 (Exercise 15.t in the learning folder).

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 adv3Lite 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 adv3Lite games (especially those created from the 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 words `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.

- `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).

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. Note that you can't set up an about box (at least, not one that uses the <ABOUTBOX> tag) if you're compiling your game for the web interface.

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 mighty look like:

```
versionInfo: GameID
    IFID = '2b5c2e11-003f-6e0c-4d75-6092f703208b'
    name = 'Bomb Disposal'
    byline = 'by Eric Eve'
    htmlByline = 'by <a href="mailto:eric.eve@whatsit.org">
                  Eric Eve</a>'
    version = '0.1'
    authorEmail = 'Eric Eve <eric.eve@whatsit.org>'
    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 *adv3Lite 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 `showCredit()`. 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 `showCredit()` 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 into 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 before 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, ... val*n*) (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). Note that it usually isn't necessary to call randomize() at the start of the game since TADS effectively does this anyway (except when running in the Workbench development/debugging environment).

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 it's better to avoid them and use the alternatives suggested below:

- Instead of **morePrompt()** use **inputManager.pauseForMore()**;

- Instead of **inputKey()** use **inputManager.getKey(nil, nil)**;

- Instead of **inputLine()** use **inputManager.getInputLine(nil, nil)**;

- Instead of **clearscreen()** use **cls()**;

The main reason for preferring these methods to the intrinsic functions is that it will make things easier if you subsequently want to convert your game to run with the WebUI interface (i.e. to make it playable remotely in a web browser).

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

As we've already seen, we can make a room dark by setting its `isLit` property to nil. If we wanted a lot of dark rooms in our game we could even define a `DarkRoom` class which did this, but for most games that would probably be overkill. Either way the effect is to create rooms that are dark unless the player manages to provide a light source. When the player character enters such a dark place, the player will see it described thus:

In the dark
It is pitch black; you can't see a thing.

If this isn't quite what we want, it's easy enough to customize. We can override the `darkName` and `darkDesc` 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: Room '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. "

   darkName = 'Cellar (in the dark)'
   darkDesc = "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
   isLit = nil
;
```

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 solution is to define `visibleInDark = true` on the staircase object; this makes the staircase visible in the dark without its providing light to see anything else by.

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 **isLit** 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 best way to test this is by using the **litWithin()** method of the Room, which will tell us if there's enough light to see by. For example, in the case of the flight of stairs leading out of the cellar, we might define:

```
+ cellarStairs: StairwayUp 'flight of stairs; dim; outline'
  desc()
  {
    if(location.litWithin)
        "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. ";
  }

  visibleInDark = true
;
```

An alternative to making **visibleInDark true** on the **cellarStairs** (or other such objects) is to use the **extraScopeItems** property to add them to scope. The standard library already puts the floor of a dark room in scope, so we don't need to do that, but we can list any other items we want added to scope in the dark:

```
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. "

    darkName = 'Cellar (in the dark)'
    darkDesc = "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

    extraScopeItems = [cellarStairs]
;
```

This behaves a little differently from making the **cellarStairs visibleInDark**. 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 see anything." 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, **extraScopeItems** 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 controlled by the `allowDarkTravel` method of the current Room, which is nil by default (meaning travel in the dark to an unlit destination is not allowed). If `allowDarkTravel` is nil, and both the current location and the potential destination are unlit, then an attempt to travel to an unlit destination will result in calling the Room's `cannotGoThatWayInDark(dir)` method, where *dir* is the attempted direction of travel (given as a direction object, e.g. `northDir`). This in turn (unless it's overridden) displays the Room's `cannotGoThatWayInDarkMsg`, which by default is "It's too dark to see where you're going. ", followed by a listing of the exits that are visible (note that `cannotGoThatWayInDarkMsg` should be defined as a singe-quoted string). We can thus override any of these methods or properties if we wish, either to allow dark-to-dark travel through a specific connector, or 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 Room
   cannotGoThatWayInDarkMsg = '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: Room
   cannotGoThatWayInDark(dir)
   {
       "Blundering around in the dark is a perilous business. You are eaten
        by a grue!<.p>";
        finishGame(ftDeath, [finishOptionUndo, finishOptionFullScore]);
   }
   isLit = nil
;
```

If you want to allow travel from a dark room in some directions but not others, you can define `visibleInDark = true` on the associated TravelConnector, for example:

```
boxRoom: Room 'Box Room'
    "There's an exit to the south and a door leads north. "
    darkDesc = "You can just about make out exits to north and south. "
    south = landing
    north = boxDoor
    isLit = nil
;

+ boxDoor: Door 'door'
    otherSide = bathDoor
    visibleInDark = true
;
```

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 **vocab** property. 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 **vocab** property directly at run-time. At least, we can change it, but doing so won't do any good, since once the game is running we're beyond the point where the library does anything with it.

The easiest way to add new vocabulary to an object is by calling its **addVocab()** method. This takes one argument, a string in the same format as that used for **vocab**. If we supply a name part to this string (any text before the first semicolon) then this will be used to change the name of the object at the same time. Otherwise the string we pass to **addVocab()** will just add some additional vocab words to the object. So, for example, calling **addVocab('; red; ball')** will add the word 'red' as an adjective and 'ball' as a noun to the vocabulary by which the object can be referred to, whereas calling **addVocab('red ball')** would additionally change the name of the object to 'red ball'. So, in the three examples we started with we might call **fox.addVocab(';
dead')**, **vase.addVocab('; broken')** and (because we now know the stranger as 'Bob') **bob.addVocab('Bob')**.

On occasion the transformation an object undergoes may be so thorough that you need to change both its name and all or most of its vocabulary. In such a case you can use the **replaceVocab()** method, which does what it says, namely replacing the original vocab string with the one supplied as an argument, and then building the name and other vocab words all over again from scratch. Thus, for example, when the ugly duckling becomes a beautiful swan you might call **duckling.replaceVocab('beautiful swan; elegant graceful; bird')**.

For most purposes **addVocab()** or **replaceVocab()** should suffice, but if you need finer control over individual words you can use **addVocabWord(word, matchFlags)** or **removeVocabWord(word, matchFlags?)**. These respectively add or remove *word* from the vocabulary that can be used to refer to the object on which they're called. The

matchFlags parameter must be one of **MatchNoun**, **MatchAdj**, **MatchPrep** or **MatchPlural** to indicate the part of speech the word we're adding or removing is meant to match. If we're adding a word, this parameter is mandatory. If we're removing a word, it's optional; if supplied the word will only be removed as that part of speech, otherwise it will be removed regardless.

10.2.2   State

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 **State** class.

When an object can be in one of several states, we can define these states as **State** objects. For example, for light sources the library defines **LitUnlit** State, while for objects that can be opened and closed there's an **OpenClosed** State. The **appliesTo(obj)** method of a State determines what objects that State is applicable to; the method should return true for any applicable *obj*. By default it does so if obj defines the **stateProp** property, where **stateProp** is a property pointer defined on the State object. For example the **OpenClosed** State defines **stateProp = &isOpen**, meaning (if **appliesTo()** hadn't been overridden) that it would have applied to every object that defines an **isOpen** property (but that would have made **OpenClosed** apply to every Thing). Nonetheless, the **OpenClosed** State still needs to know that it's associated with the **isOpen** property, just as the **LitUnlit** State needs to know it's associated with the **isLit** property, since this affects how both States behave.

This may start to become a little clearer if we show how the two State objects provided by the library are in fact defined:

```
LitUnlit: State
    stateProp = &isLit
    adjectives = [[nil, ['unlit']], [true, ['lit']]]
    appliesTo(obj) { return obj.isLightable || obj.isLit; }
    additionalInfo = [[true, ' (providing light)']]
;

OpenClosed: State
    stateProp = &isOpen
    adjectives = [[nil, ['closed']], [true, ['open']]]
    appliesTo(obj) { return obj.isOpenable; }
;
```

The **appliesTo()** method of the **LitUnlit** State means that this State is applicable to any object for which either **isLit** is true or **isLightable** is true, i.e. anything that is either lit or capable of being lit. The list in the **adjectives** property defines the

adjectives that can be applied to applicable objects when the `stateProp` takes the corresponding values. So, the first entry, `[nil, ['unlit']]` means that when `isLit` is nil on an applicable object, that object can be referred to as 'unlit'. Likewise, the second entry, `[true, ['lit']]` means that when `isLit` is true on an applicable object, that object can be referred to as 'lit'. Here an object that can be lit can be in only two states: lit or unlit, but in principle we could use the State mechanism to cater for any number of states by extending the list, e.g.:

```
Colour: State
   stateProp = &colour
   adjectives = [['red', ['red', 'scarlet']], ['blue', ['blue']],
   ['green', ['green']], ['yellow',['yellow']]]
;
```

Here we're envisaging one or more game object that have a `colour` property which can take one of four values; the Colour State would allow such objects to be referred to by their current colour (as well as described by it). As an example of why the second element in each sublist is itself a sublist, we show how red objects could be referred to as either 'red' or 'scarlet'.

To return to the `LitUnlit` State, you may have noticed that this also defines an `additionalInfo` property. This defines the additional information that should be displayed after the name of an object in certain listings (e.g. inventory lists or lists of objects in a room) when in a given state. The definition on `LitUnlit` means that the name of an object that's lit will be followed by ' (providing light)' e.g. 'a flashlight (providing light)'. So, if you want to change such messages, one way to do it would be to override the corresponding State object (i.e. `LitUnlit`, since this is the only one in the library that defines this property).

For further details, look up `State` 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 adv3Lite library provides support for this.

The most basic way of providing a light source in TADS 3 would be to change the `isLit` property of some Thing to true. So, for example, we might define:

```
magicCrystal: 'magic crystal; glowing eerie pure; light'
    "The crystal glows with a pure but eerie light. "
    isLit = true
;
```

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.

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. Adv3Lite provides the **Flashlight** class for this kind of light source. **Flashlight** inherits from **Switch** and so 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** and **isLit** properties to true. A **Flashlight** also responds to the LIGHT and EXTINGUISH commands (meaning switch on and off respectively).

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 **Container** 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: Container, Flashlight 'flashlight;;torch'

    makeLit(stat)
    {
        if(stat && !battery.isIn(self))
            "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)
    {
        if(obj != battery)
        {
            "\^<<obj.theName>> won't fit in the flashlight. ";
            exit;
        }

        else if(isOn)
```

```
    {
        /*
         * We can't use makeLit(true) here because it would have
         * undesirable side-effects.
         */
        isLit = 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, but that can be left as an exercise for the reader. In essence we'd need some kind of Daemon to reduce the battery power each turn the flashlight was on until the battery was drained (or we could use the FueledLightSource extension).

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, each of which only stays lit for a couple of turns before burning out. There's a candle in the first cave (but only with a limited life). Another cave contains an oil lamp, but it's low on oil. Yet another cave contains an oil can you can refill it from, and another cave contains a flashlight. The final cave contains a rusty old box containing the crystal.

11   Nested Rooms

11.1   Nested Room Basics

Back in chapter 5 we discussed the containment model in adv3Lite, 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; or rather they don't provide any handling for the player character and other actors to get in and out of them.

It doesn't take much to change that in adv3Lite. In order to make an object an actor can get on, all we need to do is to give it a `contType` of `On` and then define `isBoardable = true`. In most cases, though, it will be easier just to use the `Platform` class, which does all that for us (a Platform is a `Surface` with `isBoardable = true`). Similarly to make something an actor can get in, we just need an object with `contType = In` and `isEnterable = true`, and again there's a `Booth` class that does that for us (a `Booth` is a `Container` with `contType = In`).

Platforms and Booths are objects that aren't rooms but which can contain an actor; they're typically used to implement items of furniture and the like. Getting on or off a Platform or in or out of a Booth doesn't count as travel and doesn't trigger any travel notifications. If the player types OUT when the player character is on a Platform or in a Booth, it's taken as a command to get off or out.

For the most part, we can use Nested Rooms 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'
;

+ bed: Platform, Heavy 'single bed'
;

+ woodenChair: Platform 'wooden chair; small'
  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 get on the chair, or on the bed. 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.

That's *almost* all you need to know about Nested Rooms in adv3Lite; but not quite. In the remainder of the chapter we'll look at some of the potential complications.

The first one is this: what happens when there's something you want to be able to both get on and get in? Well, that depends on what exactly you have in mind. If what you want is a bed, say, that responds to GET ON BED or GET IN BED in exactly the same way, then you could simply define the bed as a Platform and then make GET IN behave like GET ON and GET OUT like GET OFF using the **asDobjFor()** macro, like this:

```
bed: Platform, Heavy 'bed'
   dobjFor(Enter) asDobjFor(Board)
   dobjFor(GetOutOf) asDobjFor(GetOff)
;
```

If, on the other hand, what you have in mind is something like a large cabinet the player character could either squeeze inside or get on top of, then you need to use the kind of multiple containment techniques we met in chapter 5, for example:

```
cabinet: Heavy 'tall metal cabinet; green'
    "It's just tall enough for you to squeeze inside, but not so tall
     that you couldn't climb on top of it. "
    remapOn: SubComponent { isBoardable = true }
    remapIn: SubComponent { isEnterable = true }
;
```

11.2   Nested Rooms and Postures

Adv3Lite makes no attempt to track the postures of actors[1], so STAND ON BED, SIT ON BED and LIE ON BED all mean the same as GET ON BED, just as SIT IN CABINET, LIE IN CABINET, and STAND IN CABINET would all mean the same as GET IN CABINET.

That's true, at least, unless the game author wants to make a slight distinction between them. While the *effects* of STAND ON BED, SIT ON BED, LIE ON BED and GET ON BED are all the same, we can, if we wish, treat them slightly differently in terms of which we're prepared to allow, and which we may prefer. A small wooden chair is probably something the player character couldn't actually lie on, so although in one

---

1   If you really need to track the postures of actors, there's a postures.t extension that comes in the extensions directory of your adv3Lite installation; but for most games this is unlikely to be either necessary or beneficial.

sense LIE ON CHAIR means the same as GET ON CHAIR, we might just want to disallow the former and allow the latter for the sake of slightly greater realism. If we want to do this, we can do so by setting `canLieOnMe` to nil (and likewise for `canStandOnMe` and `canSitOnMe`). The effect is a subtle one, but it may be worthwhile.

It's also one that may have a desirable knock-on effect. If the player types the command **lie down**, since adv3Lite provides no means for the player character to change posture, the command will be treated as meaning **lie on** with a missing direct object. The game will then prompt the player for the missing direct object ("What do you want to lie on?") *unless* there's one object in scope that scores better for this action than any other. If we rule out lying down on the chair, then the bed is the only candidate, so **lie down** will be taken to mean **lie on bed** without the player being prompted for the missing direct object.

But what happens if the player types **sit**? Presumably it should be possible to sit on either the bed or the chair, but other things being equal we might think the chair is the better choice. We can signal this by giving it a higher `sitOnScore` than the bed (the default being 100; we could also use `standOnScore` and `lieOnScore` in the same way).

So, with these minor refinements, our little bedroom might become:

```
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'
;

+ bed: Platform, Heavy 'single bed'
;

+ woodenChair: Chair 'wooden chair; small'
  initSpecialDesc = "A small wooden chair next to the bed takes up most
  of the spare space in the room. "

  canLieOnMe = nil
  sitOnScore = 120
;
```

11.3   Other Features of Nested Rooms

11.3.1   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.

Since by default everything has a `bulk` of 0 and a `bulkCapacity` of 10000, this won't present any limitations unless you start defining the `bulk`, `bulkCapacity` and `maxSingleBulk` of various objects. In the case of nested rooms you may wish to do so, for example, if you want a chair that's just big enough for one person to sit on. Giving the chair the same bulkCapacity as the bulk you assigned to actors would then prevent the player character from sitting on Aunt Florence's favourite armchair while it was occupied by Aunt Florence. It could also be used to prevent the player from getting in a trunk that was already filled with junk, for example.

11.3.2   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 large rug is likely to result in the object's falling onto the Platform. Dropping something while on a small chair is likely to result in the object's falling into the chair's enclosing room.

Which of these is chosen depends on the value of the `dropLocation` property of the actor's immediate container. By default this is `self`, meaning that an object dropped while an actor is in a nested room will end up in or on that nested room (Booth or Platform). For a small nested room like a small chair, you could change this so that, for example, the `dropLocation` is the location of the chair, for example:

```
+ woodenChair: Chair 'wooden chair; small'
  initSpecialDesc = "A small wooden chair next to the bed takes up most
  of the spare space in the room. "

  canLieOnMe = nil
  sitOnScore = 120
  dropLocation = location
;
```

Note that you can also define the `dropLocation` property on a Room, if you wish. This might be useful if you have a room representing a space well off the ground, such as the top of a tree or the top of a mast. You could then define the dropLocation to be the ground or deck below, so that items dropped at the top of the tree or mast ended up falling to the room below and rather than suspended in mid-air.

11.3.3   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 transparent, s/he will not be able to see out into the enclosing room. In such a case the Nested Room's `roomTitle` property will be used as the name of the player character's current location, and the `interiorDesc` 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 Booth with no source of light, he won't be able to see anything at all; but we can then override the `darkName` and `darkDesc` properties of the Booth to provide custom versions just as we can for a Room. By default, the `roomTitle` of a Nested Room is simply its name.

11.3.4   Staging and Exit Locations

If the player character is on a bed and wants to get on a chair elsewhere in the same room, the command **get on chair** shouldn't just teleport him from one nested room to the other. The player character first needs to leave the bed before s/he can get on the chair. To enforce this adv3Lite uses the `stagingLocation` property, which contains the location an actor needs to be in in order to enter or board the nested room in question. For most nested rooms the `stagingLocation` will simply be the same as the location, although for nested rooms that are SubComponents of some multiply-containing objects the `stagingLocation` will be the lexicalParent's location, i.e. the location of the mutliply-containing object.

The rule that an actor must be in a nested room's `stagingLocation` before entering or boarding it is enforced in the `actorInStagingLocation` precondition, which will attempt to move the actor into the staging location via one or more implicit actions. Note that this is only enforced if the actor is outside the nested room. If the actor is on a chair that's resting on a stage, for example, then **get on stage** will not trigger an attempt to move the actor to the stage's `stagingLocation`; the actor will simply be moved straight to the stage.

The converse of the `stagingLocation` is the `exitLocation`, which is where the actor is moved to in response to a command to **get off** or **get out of** the object in question. The `exitLocation` is also defined by default to be either the object's location or, if appropriate, the location of its lexicalParent, whereas by default the `stagingLocation` is the same as the `exitLocation`.

11.4   Vehicles

Occasionally you may want a nested room the player character can move around in or on, like a bicycle, a magic carpet or a cart. Such objects are often called *vehicles* in Interactive Fiction. To create a vehicle in adv3Lite you just define a Booth or Platform in the normal way and then define `isVehicle = true` on it. For example, to make a basic bicycle you might do this:

```
bike: Platform 'bicycle; old battered; bike'
   "It's your old battered bike, still serviceable though. "
   isVehicle = true
   canLieOnMe = nil
   canStandOnMe = nil
;
```

The player can now issue the commands **get on bike** or **sit on bike** to get the player character on the bicycle. Once the player character is on the bike, any movement commands like **north** or **climb stairs** will move the bicycle with the player character on it.

That, of course, raises the question whether the player character *ought* to be allowed to climb the stairs while riding the bike. The solution to this potential problem lies in the fact 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. We could thus define a **bikeBarrier TravelBarrier** to put on any TravelConnectors we don't think the bike should be allowed to traverse.

```
bikeBarrier: TravelBarrier
  canTravelerPass(traveler, connector) { return traveler != bike; }
  explainTravelBarrier(traveler, connector)
  {
       "{I} {can\'t} ride the bike <<connector.traversalMsg>>. ";
  }
;
```

This, incidentally, illustrates why the TravelBarrier class takes a *connector* parameter in its methods. Since our bikeBarrier could be used on any number of TravelConnectors, we need to be able to phrase the message in **explainTravelBarrier()** in such a way that it will suit any one of them. The *connector* parameter of **explainTravelBarrier** gives us the TravelConnector that's being attempted, and its **traversalMsg** property gives us a snippet of text like 'up the staircase' or 'through the front door' that can be used to complete a sentence of the type shown above.

11.5   Reaching In and Out

The normal convention in Interactive Fiction is that everything in the current room is within reach of an actor located there, possibly on the assumption that the actor will unobtrusively move round the space delineated by the 'room' to interact with whatever's needed. This assumption becomes a bit less plausible, however, if the actor is located in a nested room, sitting on a chair, for example. In such a case, one of three things could happen. First, if the player character needs to reach an object that's across the room we could just ignore the lack of realism involved and let the player character take it without moving from the chair. Second, we could block the player character's access to objects outside the chair by saying something like "You can't reach the green book from the chair." Or third, we could enforce the rule that the player character can't reach across the room from the chair, but automate making the player character leave the chair in order to reach an object placed elsewhere in the room.

None of these solutions is obviously the 'right' one to be preferred above the others, and adv3Lite allows us to choose which we want to use in any given case. We can do so by taking advantage of the following properties/methods (which we would need to defined on the chair or other nested room the player character might be occupying): **allowReachOut(obj)** and **autoGetOutToReach**. If **allowReachOut(obj)** returns true, then an actor on the chair or other nested room can reach out to touch *obj* without leaving the nested room; if it returns nil then s/he can't. This allows us to treat different objects differently, depending on how we envisage them being spatially related to the nested room. Note that adv3Lite assumes that an actor can always reach the nested room or objects inside the nested room in any case, so we don't need to cater for such obviously accessible objects in our **allowReachOut(obj)** method. If **allowReachOut(obj)** returns nil, then if **autoGetOutToReach** is true, the actor will automatically leave the nested room in order to reach *obj*; otherwise if **autoGetOutToReach** is nil the attempt to reach obj will fail with a suitable message explaining that *obj* can't be reached.

So, for example, if the player character starts out on bed and we want him/her to have to leave the bed in order to touch anything else in the room we might do something like this:

```
me: Thing
    location = bed
    isFixed = true
    person = 2
    contType = Carrier
;


bedroom: Room 'bedroom'
  "This is your bedroom..."
;


+ bed: Platform, Heavy 'bed'
    allowReachOut(obj) { return nil; }
    autoGetOutToReach = true;
;

+ table: Surface 'table'
;

++ note: Thing 'note'
   "You suspect it may be important, but you'd need to read it
    to be sure. "

   readDesc = "You have five minutes to get out of here before
     this house burns down around you! "

   dobjFor(Read)
   {
       preCond = [objHeld]
   }
;
```

In this case the player can see that there's an important note on the table, but the player character can't actually read the note without holding it, and to do that s/he needs to leave the bed. In this case an attempt to read the note will trigger an implicit take action which in turn will trigger an attempt to get off the bed in order to reach the note.

The complement to reaching out is reaching in, and we can also control that, or indeed, just reaching. This may or may not involve nested rooms, and in any case works a little differently from the **allowReachOut** mechanism we have just seen. Reaching and reaching in are controlled by two methods, **checkReach(actor)** and **checkReachIn(actor, target)**. The **checkReach(actor)** method determines whether *actor* can reach the object **checkReach()** is defined on; **checkReachIn(actor, target)** determines whether *actor* can reach inside the object on which it's defined to reach *target*. By default **checkReachIn(actor, target)** simply calls **checkReach(actor)**, on the assumption that if an actor can't reach an object, that actor will also be unable to reach that object's contents.

Unlike **allowReachOut()**, these two methods should not return true or nil; instead they should either do nothing or else display a message explaining why the object on which they're defined can't be reached. If they display anything it will be assumed that reaching is not possible; if they don't then it will be assumed that they're not objecting to the reach.

This mechanism can be used for a number of purposes, for example to make an object too hot to touch (perhaps until it cools down) or else out of reach on a high shelf (until the player character gets on a chair to reach it, say).

We could implement the second of these scenarios like so:

```
+ Surface, Fixture 'high shelf'
   checkReach(actor)
   {
      if(!actor.isIn(woodenChair))
         "The shelf is too high up for you to reach. ";
   }
;

++ torch: Flashlight 'flashlight;; torch'
;

+ woodenChair: Platform 'wooden chair'
;
```

In this example, there's a flashlight (which the player presumably needs in order 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 previously indicated, we're not restricted to using this mechanism for situations where an object may be too far away; we can use it for *any* situation in which an object might become untouchable. For example, suppose we have an iron poker which we can put in a fire, which gets hotter the longer we leave it there, and which eventually becomes too hot to touch. Rather than trying to trap every action that involves touching the poker, we can simply make it out of reach with the `checkReach()` method:

```
poker: Thing 'iron poker'
  "It's <<isRedHot ? 'glowing red hot' : 'just an iron poker'>>. "
  isRedHot = nil

  checkReach(actor)
  {
    if(isRedHot)
      "It's too hot to touch! ";
  }
;
```

All the methods and properties we have defined in this section are defined on Thing, so they're not restricted to being used with nested rooms.

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 Nested Room can't reach outside it (there will need to be exceptions to this), and will automatically be taken out of the Nested Room if s/he tries.

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. On analogy with things we've covered previously, you may suppose that we can do this by simply defining `isLockable = true` on an object. But it's actually a bit more complicated than that, since we need to determine not only whether something is lockable, but if so how. So there is, in fact no `isLockable` property in adv3Lite. Instead, there's a `lockability` property which can take one of four values:

- `notLockable` – this object can't be locked and unlocked at all (this is the default)

- `lockableWithoutKey` – this object can be locked and unlocked using the **lock** and **unlock** commands without the need of any key or any other mechanism (this might represent a door with a simple paddle lock, for example).

- `lockableWithKey` – this object can be locked and unlocked with a suitable key.

- `indirectLockable` – this object can be locked and unlocked, but only by means of some external mechanism (such a pulling a lever or pushing a button or entering a code on a keypad).

The first two cases are completely straightforward and don't really require any further comment, beyond the fact that in all four cases whether something is currently locked is determined by the value of its `isLocked` property, and that to lock or unlock something programmatically you should call its `makeLocked(stat)` method (with *stat* = true to lock and nil to unlock). If this is called on one side of a door the Door class will ensure that the other side is kept in sync.

The third case is more complex, however, since we need to determine which keys fit which locks. This is determined, not by the lockable objects, but by the keys that can lock and unlock them. A key must be of the `Key` class, and can define one or more of the following properties in relation to what it can lock and unlock:

- `actualLockList` – a list of the objects this key can in fact lock and unlock; this must be defined if the key is to be of any actual use. Note that if a key works on both sides of a door, both sides of the door must be included in this list.

- `plausibleLockList` – a list of the other objects this key looks like it might be able to lock and unlock (because the lock is of the same size and type). You don't have to define this but it can make the parser behave more intelligently if

you do, for example by not attempting to unlock a Yale lock with a card key.

- **knownLockList** - a list of the objects the player character knows that this key can lock and unlock. This is maintained automatically by the library in that once a key has successfully been used to lock or unlock something, the object just locked or unlocked is added to this list; but game code may occasionally also want to define this for keys the player character would start out knowing about, such as his/her own front door key.

- **notAPlausibleKeyMsg** – a single-quoted string giving the message to display if the player attempts to use this key on a lock it obviously won't fit (i.e. an object that's in neither the **actualLockList** nor the **plausibleLockList**).

- **keyDoesntFitMsg** – a single-quoted string giving the message to display if the player attempts to use this key on a lock it in fact doesn't fit.

So, for example, to define a key that in fact locks and unlocks the front door of a house but looks as it might also work on the back door we might write:

```
brassKey: Key 'chunk brass key'
   actualLockList = [frontDoorInside, frontDoorOutside]
   plausibleLockList = [backDoorInside, backDoorOutside[]
;
```

Since containers with locks are relatively common the library defines the **LockableContainer** class to mean an openable container with **lockability** = **lockableWithoutKey**, and the **KeyedContainer** class to mean an openable container with **lockability** = **lockableWithKey**.

This leaves only the fourth case, objects with a **lockability** of **indirectLockable**. These are objects (such as containers and doors) that can be locked and unlocked, but which need some kind of external mechanism to lock or unlock them. We can't generalize about this case, except to say that any external mechanism would need to call **makeLocked(nil)** on such an object to unlock it, and **makeLocked(true)** to lock it. In the next section we'll look at some of the gadgets and devices that could be used to lock and unlock indirectly lockable items, as well as for a number of other purposes. In the meantime the one other thing to note about indirectly lockable objects is that any attempt to lock or unlock them via a **lock** or **unlock** command will result in nothing happening beyond a display of their **indirectLockableMsg**, which is defined as a single-quoted string.

12.2   Control Gadgets

As we've just seen, if we define something (typically a door or container) with a **lockability** of **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**, **Switch**, **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 **makePushed()** 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: Door 'door'
   indirectLockableMsg = 'Maybe that's what's the brown button is for. '
;

+ Button 'brown button; large'
    makePushed()
    {
      "A loud <i>click</i> comes from the door in the panelling. ";
      panelDoor.makeLocked(!panelDoor.isLocked);
    }
  }
;
```

A Button is assumed to be fixed in place by default, so there's no need to make it a Fixture too.

A **Lever** is slightly more complicated in that it has two states, pulled or pushed (determined by the value of its **isPulled** and **isPushed** properties; by default **isPushed** is simply the opposite of whatever **isPulled** is, but we define both properties in case we want to override this to make a lever that can be in more than two states. When **isPulled** is true the player has to **push** the lever to move it; when **isPushed** is true 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: Door 'secret door;; panel'
   isHidden = isOpen
;


+ desk: Heavy 'oak desk; wooden'
   remapOn: SubComponent { }
   remapUnder: SubComponent
   {
      dobjFor(LookUnder)
      {
         action()
         {
            if(hiddenLever.isHidden)
            {
              "You find a small concealed lever fixed to the underside of the
               desk. ";
               hiddenLever.discover();
            }
         }
      }
   }
;


++ hiddenLever: Lever 'small lever; hidden concealed'
   subLocation = &subUnderside

   isHidden = true

   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 slightly different kind of control is the `Switch`, which, as its name suggests, is a control the player can **turn on** or **turn off** (or **switch** on and off). A Switch has an `isOn` property that determines whether it's on or off. Changing a Switch 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 `Switch` do something interesting (in fact, both these properties/methods are defined on Thing).

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:

```
+ Switch, Fixture 'light switch'
    makeOn(val)
    {
        inherited(val);
        lightBulb.makeLit(val);
        "The light bulb <<val ? 'comes on' : 'goes out'>>. ";
    }
;
```

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 `Switch` suitably defined 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 adv3Lite 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* converted to lower case, to make it easier to test whether or not it's a valid setting for this item.

- `validSettings` – a list of the settings that are valid for this Settable item, expressed as a list of singe-quoted strings, normally in lower case.

- `isValidSetting(val)` – by default this returns true if and only if *val* converted to canonical form by `canonicalizeSetting()` is among the values listed in the `validSettings` property, but we can override this to follow some other rule if we wish (as `NumberedDial` does; see below).

- `makeSetting(val)` – this is the method that changes the setting, by changing `curSetting` to *val*. Note that when this is called *val* has already been converted to canonical form by `canonicalizeSetting()`. This is probably the most convenient method to override if you want changing the setting of this item to have any interesting effect, but if you do override it, remember to call `inherited(val)` in your overridden version.

Two further properties of possible interest are `okaySetMsg` and `invalidSettingMsg`

which can be overridden to 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. For a labeled slider we can simply define:

```
+ Settable 'slider'
   "The slider can be set <<orList(validSettings)>>. It's currently set to
   <<curSetting>>. "
   validSettings = ['red', 'yellow', 'blue' ]

   makeSetting(val)
   {
      inherited(val);
      if(val == 'red')
        "A klaxon starts to sound. ";
   }
;
```

If we're implementing a Settable that isn't a dial (it's a slider, say), but we want to be able to set it to one of a range of numbers, the simplest thing to do is to "borrow" the `isValidSetting()` method from the `NumberedDial` class (which we'll meet shortly below). Or we might simply *use* this class 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" the method we could define a numbered slider like this:

```
+ Settable 'slider'
    "It can be set to any number between <<minSetting>> and <<maxSetting>>.
     It's currently set to <<curSetting>>. "
    minSetting = 0
    maxSetting = 70
    curSetting = '60'

    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! ";
    }
;
```

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' literalDobj
     :
;
```

A specializiation of **Settable** is the **Dial** class, which simply allows **turn dial to x** as well as **set dial to x**. Otherwise it behaves in exactly the same way.

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 '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.

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.

13   More About Actions

13.1   Messages

As we've seen, it's possible to use macros like `illogical('You can't do that. ')` in verify routines, but it's more usual to see them specified in the form `illogical(cannotTakeMsg)`, where `cannotTakeMsg` is a property of Thing. This makes it easy to override properties like `cannotTakeMsg` with a single-quoted string of your own devising to provide your own custom response to taking something that can't be taken, without having to override a complete verify method to do so. The same principle is followed for other kinds of responses too. The name of the message can often be guessed from the name of the action. Thus, if the action was called `Foo`, a message ruling it out at the verify stage would probably be called `cannotFooMsg`. For an action that takes two objects, a TIAction that is, the message will probably be called `cannotFooMsg` on the direct object and `cannotFooPrepMsg` on the indirect object, where Prep is the preposition that forms part of the action name (e.g. With if the action were called `FooWith`).

We can't list all the adv3Lite library messages here, but if you need to find out where one is defined or what it is called there are various options available to you. One is to click on the Messages tab of the *adv3Lite Library Reference Manual*. This will provide you with an alphabetical list (in a somewhat special sense of 'alphabetical') of library messages which your web browser should allow you to search. Another place to look is in the file tads3Lite_msgs.csv which is located in the manual folder of your adv3Lite documentation. You can open this file in a spreadsheet program such as Microsoft Excel and then sort it and search it in any way you like. A third way is to use the Action Reference chapter of the *adv3Lite Manual*; you can use this to look up a particular action and then see some of the principal message properties associated with that action.

If you do look up any of the messages defined in the library, you'll see that they're defined with either the `BMsg()` or the `DMsg()` macro, and tend to take the following form:

```
BMsg(message name, 'message text', params...)
DMsg(message name, 'message text', params...)
```

The difference between them is that BMsg (Build Message) returns a singe-quoted string containing the message text, while DMsg (Display Message) displays the message straight away.

The 'message text' part of these definitions can just be a single-quoted string containing the text to be displayed, but it can also contain a number of items in curly braces, like `{The subj dobj}` or `{1}`. The numbers in curly braces refer to the corresponding parameter in the (optional) list of params that follows, so that, for

example:

```
DMsg (eat something, 'You eat {1} with {2}. ', food.theName, 'relish');
```

Would result in the display of "You eat the strong cheese with relish. " if food is an object whose theName is 'the strong cheese'.

The other items in curly braces are called *message parameter substitutions*, and can be used virtually anywhere in a double- or single-quoted string. These are pieces of placeholder text that are replaced with particular words according to context when the string containing them is displayed, allowing the containing strings to adapt to the circumstances of their use.

Commonly used message parameter substitutions include:

- **{The subj obj}**; the theName of obj, marked as the subject of the next verb to follow.

- **{A subj obj}**; the aName of obj, marked as the subject of the next verb to follow.

- **{the obj}**; theName of obj, regarded as the object of a verb.

- **{her obj}**; the possessive form of obj (e.g. "Brian's", or "my", or the "baker's")

- **{himself obj}**; the reflexive form of obj (e.g. "herself", "himself", or "yourself")

- **{i}**; the current actor, in the subjective case (e.g. "I", "you", "Bob", or "the baker")

- **{me}**; the current actor, in the objective case (e.g. "me", "you", "Bob", or "the baker")

- **{my}**; the possessive of the current actor (e.g. "my", "your", "Bob's")

- **{that obj}**; 'that' or 'those' (depending on whether obj is singular or plural), regarded as the object of verb.

- **{that subj obj}**; 'that' or 'those' as the subject of the following verb.

- **{is} {am}** or **{are}**; the verb 'to be' in agreement with the most recent subject; in the absence of such a subject, the subject is assumed to be in the third person singular.

- **{come}**; the verb 'to come' in agreement with the most recent subject, and so on for about two hundred other irregular verbs.

- **{dummy}**; a dummy subject marker that displays nothing, but makes any verbs that follow agree with a third-person singular subject.

- **{plural}**; a dummy subject marker that displays nothing, but makes any verbs

that follow agree with a third-person plural subject.

- `love{s/d}`; the verb 'to love' in agreement with the most recent subject, and so on for all the other regular verbs that follow this pattern.

- `frown{s/ed}`; the verb 'to frown' in agreement with the most recent subject, and so on for all the other regular verbs that follow this pattern.

- `splash{es/ed}`; the verb 'to splash' in agreement with the most recent subject, and so on for all the other regular verbs that follow this pattern.

- `cr{ies/ied}`; the verb 'to cry' in agreement with the most recent subject, and so on for all the other regular verbs that follow this pattern.

- `stop{s/?ed}`; the verb 'to stop' in agreement with the most recent subject, and so on for all the other regular verbs that double their final consonant in the past tense.

For a complete list see the chapter on Messages in the *adv3Lite Library Manual*.

Where *obj* appears in any of these it can be one of:

- `dobj` – the direct object of the current action

- `iobj` – the indirect object of the current action

- `cobj` – the current object of the current action (which may be either the direct or the indirect object, depending on context)

- `actor` – the current actor

- The `globalParamName` of an object (defined by giving it a globalParamName property, defined as a single-quoted string)

- A temporary message parameter `val` defined by the `gMessageParams(val)` macro where *val* is the name of a local variable.

Note that where a message parameter substitution starts with a capital letter, so will the text that's substituted for it, so we normally want to start a message parameter substitution with a capital letter if it appears at the start of a sentence, but not otherwise, so:

`"{I} {cannot} {see} {the dobj} {here}. ";`
But

`"It {dummy} seem{s/ed} that {i} {cannot} {see} {the dobj} {here}. ";`

(`{here}` is a message parameter substitution that becomes 'here' in the present tense and 'there' in the past tense).

But to get back to `DMsg()` and `BMsg()`, we still haven't explained the message name that appears in formats such as:

`Dmsg(cannot see obj, '{I} {cannot} {see} {the dobj} {here}. ');`

In this instance, `cannot see obj` is the message name, and it can be used if we want to change the message to something else, either because we're translating the library wholesale into another language, or because we want to customize the library's messages. To customize library messages we need to define a `CustomMessages` object. This contains a list of `Msg()` macros which in turn give the message name followed by the message string we want to replace it with, for example:

```
CustomMessages
    messages = [
        Msg(cannot see obj, '{The subj dobj} {isn\'t} anywhere to be seen. ')
    ]
;
```

Or, to give a more extensive example, this is how we might customize the library messages associated with the Take action:

```
CustomMessages
    messages = [
        Msg(report take, 'Snatched. | You grab {1}. '),
        Msg(fixed in place, 'Idiot; any fool can see {the subj dobj} {is} firmly
            nailed down. '),
        Msg(already holding, 'In case you hadn\'t noticed, you\'re already
            holding {the dobj}. '),
        Msg(cannot take my container, 'Great idea. Just how do you propose to
            pick up {the dobj} while you\'re right {1}? ')

    ]
;
```

Note that in such cases we can't change the parameters that place-holders like `{1}` refer to; they continue to retain the same meaning they had in the original `BMsg()` and `DMsg()` definitions. We can, however, use the place-holder mechanism to construct strings outside the BMsg()/DMsg() mechanisms by using the `dmsg()` and `bmsg()` functions; e.g.:

```
dmsg('The {1} in {2} falls mainly on the {3}. ', 'rain', 'Spain',
'tourists');

cannotBendMsg = bmsg{'{I} {can\'t} bend {1}. ', theName)
```

Finally, we there's a couple of properties of `CustomMessages` we can use to determined which `CustomMessages` object to use. The `priority` property determines which CustomMessages object takes precedence if more than one overrides the same message; the higher the priority the higher the precedence (to override library messages use a `priority` of 200 or above). The `active` property determines whether the CustomMessages object will be used at all; you could switch this between true and nil, for example, to change the tone of your game between different narrators.

13.2   Stopping Actions

We have already encountered the **exit** macro, which can be used to stop an action in its tracks. 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 (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.

Slightly different is the effect of the **abort** macro. This not only stops the current action, but skips over the iteration of the action on any other objects entered on the command line at the same time and also skips over any end of term processing like Fuses and Daemons. It does not, however, cancel any separate commands entered on the same line.

To illustrate the difference, suppose we define the following:

```
streetCorner: Room, CyclicEventList 'Street Corner'
    "The corner of the street. A store lies to the south. "

    south = store

    eventList = [
        'A gust of wind blew. ',
        'A cloud passed over the sun. ',
        'The sun came out again. '
    ]
;

+ ball: Thing 'ball'

    beforeAction()
    {
        if(gDobj == ball)
        {
            "The ball wouldn't allow that. ";
            exit;
        }
    }
;

+ bat: Thing 'bat'
;
```

```
store: Room 'store;; shop'
   "The way out is to the north. "
   north = streetCorner
;
```

In this case if we enter the command **take ball and bat**, the `exit` statement will prevent the ball being taken, but not the bat, and we'll still see one of the atmospheric messages like 'A gust of wind blew.' If we changed `exit` to `abort` then we'd just see the refusal to do anything with the ball; there'd be no attempt to take the bat and no atmospheric message. If we had entered the command **take ball and bat; throw bat; go south** however, then the commands **throw bat** and **go south** would be executed even with `abort`, but we'd only see one atmospheric message instead of the two we would have seen had we used `exit` rather than `abort`. In other words, each separate command entered on the command line is treated as separate; `abort` completely stops the current command (but not any subsequent commands entered on the same command line), whereas `exit` merely terminates the execution of the current action with the current object.

Both `exit` and `abort` are in fact macros that throw Exceptions. 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 `abort` 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 abort throw new AbortActionSignal()
```

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 `doSomeStuff()` 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 *adv3Lite 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: Actor 'Bob;;man; him'

    beforeAction()
    {
        inherited;
        if(gActionIs(Yell))
        {
            "There\'s no need to shout; Bob can hear you
               perfectly well. ";
            exit;
        }
    }
;
```

If we want the player's *location* to intervene, however, we should use its **roomBeforeAction()** method; e.g.:

```
lowCave: Room 'Low Cave'
   "There's not much headroom here. "
   roomBeforeAction()
   {
      if(gActionIs(Jump))
      {
        "You\'d better not jump here; you\'d bump your head on the
          low ceiling. ");
       exit;
      }
   }
;
```

We could do the same thing for an entire Region by using **regionBeforeAction()**, for example:

```
caveRegion: Region
   regionBeforeAction()
   {
      if(gActionIs(Jump))
      {
        "You\'d better not jump here; you'd bump your head on the
         low ceiling. ");
         exit;
      }
   }
;
```

When exactly these methods are run depends on the value of an option set in **gameMain.beforeRunsBeforeCheck**. If this is true then **roomBeforeAction()**, **regionBeforeAction()** and **beforeAction()** run between the verify and check stages of the action (in that order). If it is nil (the current default), 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(), roomAfterAction()** and **regionAfterAction()** methods. 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'
  "It looks very delicate. "
  afterAction()
  {
    if(gActionIs(Yell) && !vase.location.ofKind(Actor))
      "The vase vibrates alarmingly. ";
  }
;

+ bob: Actor 'Bob;; man; him'

   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()`, `regionBeforeAction()` and `beforeAction()` are called (in that order), `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))
        {
          "You can't stand up while you're all tied up. ";
          exit;
        }
    }
    isTiedUp = nil
;
```

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 who's 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 TravelConnector (which calls `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: Room 'Bank of River' 'bank of the river'
   "A narrow bridge spans the river to the north. "
    north = bridge
;

+ troll: Actor 'troll; mean mean-looking; beast; him'
   "A truly mean-looking beast! "
   beforeTravel(traveler, connector)
   {
      if(traveler == me && connector == bridge)
      {
         "The troll blocks your path with a menacing growl! ";
         exit;
      }
      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 so that omitting it 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 TravelConnector to its **north** property:

```
riverBank: Room 'Bank of River' 'bank of the river'
   "A narrow bridge spans the river to the north. "

   north: TravelConnector
   {
      destination = bridge
      travelDesc = "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: Actor 'troll; mean mean-looking; beast; him'
   "A truly mean-looking beast! "
   beforeTravel(traveler, connector)
   {
      if(traveler == me && connector == riverBank.north)
      {
         "The troll blocks your path with a menacing growl! ";
         exit;
      }
      inherited(traveler, connector);
   }
;
```

This will then work whether `riverBank.north` is left as `bridge` or subsequently changed to a TravelConnector.

The most convenient travelling equivalent of `roomBeforeAction()` is `travelerLeaving(traveler, dest)`, which would need to be overridden on the Room the *traveler* (actor or vehicle) is about to leave to go to *dest*. We can also define this method on a Region, in which case it will be triggered when the traveler is about to move from a room that's in the Region in question to one that isn't.

All the travel methods we have just discussed are called on the region, 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 region, 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: Actor 'Bob;; man; him'
   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 `travelerEntering(traveler, origin),` where the *origin* parameter is the room *traveler* is about to travel from. There's also a `travelerEntering(traveler, origin)` method of Region, where *origin* is once again the room traveled from. This method is only called on a Region after *traveler* has entered a room that's in the region from a room that isn't.

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 may be a bit of a problem. One workaround would be to use the `travelerEntering()` method to set up a Fuse to be executed at the end of the same turn:

```
store: Room 'Store'
    "All sorts of interesting goods are on sale here. The way out
     is back to the south. "

    south = streetCorner

    travelerEntering(traveler, origin)
    {
        "{I} enter{s/ed} from <<origin.theName>>. ";
        new Fuse(self, &enterMsg, 0);
    }

    enterMsg = "{I} look{s/ed} around the shop with keen interest. "
;
```

If you just want to display a message after entering a room, however, it's probably simpler just to use afterTravel():

```
store: Room 'Store'
    "All sorts of interesting goods are on sale here. The way out
     is back to the south. "

    south = streetCorner

    afterTravel(traveler, connector)
    {
        "{I} look{s/ed} around the shop with keen interest. ";
    }
;
```

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 adv3Lite, 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'
    dobjFor(Take)
    {
        check()
        {
            if(gActor.isPlayerChar())
                "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 vase; cut glass'
    okayDropMsg = 'You carefully lower the vase to the ground. '
;
```

We must write:

```
vase: Container 'priceless vase; cut glass'
    okayDropMsg = '{I} carefully lower{s/ed} 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.

To make an NPC carry out an action we can use `replaceActorAction()` or `nestedActorAction().` We could use either of these to make an NPC carry out a brand new action; the only difference is that `replaceActorAction()` adds an exit statement after executing the action to stop the action it was called from, while `nestedActorAction()` would allow any calling action to continue (if, for example, either macro were called from the action routine of some other action). Both functions 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:

```
nestedActorAction(bob, Jump);
nestedActorAction(bob, Take, redBall);
nestedActorAction(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 adv3Lite.

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. Adv3Lite 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. Rather than try to cover every aspect of it, the present chapter will simply try to give an overview of what is possible with NPCs in adv3Lite, 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 adv3Lite 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

The **Actor** class defines most of the behaviour needed for animate objects (NPCs). NPCs defined with the **Actor** class are non-portable by default (so the player character can't pick them up and take them around). If you need to define any Actor that are portable, perhaps small animals such a cats, mice and rabbits, you could override the **isFixed** property to nil on them (though often the behaviour of such small animals in a work of IF won't really be complex enough to require the use of the Actor class).

The definition of an Actor object can be fairly minimal: we need to specify its **vocab,** including its **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 including 'him' or 'her' at the end of its vocab string. If the Actor has a proper name (e.g. 'Bob' rather than 'the tall man' or whatever), then so long as every word in the name section of the vocab string starts with a capital letter, the name will be treated as a proper name (i.e. the library won't precede it with 'a' or 'the' when referring to it), although if we do define what's meant to be a proper name that includes some words not beginning with a capital letter we would need to explicitly define **proper = true** on the Actor. Thus a minimal Actor object definition might look like:

```
mavis: Actor 'Aunt Mavis; old frail; woman; her'
   "Well past her prime, she is now looking distinctly frail. "
;
```

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: Actor 'Aunt Mavis; old frail; woman; her' @lounge
   "Well past her prime, she is now looking distinctly frail. "

   shouldNotKissMsg = 'She\'s not of a generation that welcomes outward
      shows of affection. '
   cannotEatMsg = 'Whatever else Aunt Mavis is, she is definitely not that
      tasty. '
   shouldNotAttackMsg = '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 subj mavis}'**, 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;;; him' @highStreet
    "He's a tall man, wearing a smart business suit and sporting a thin
      moustache. "

    globalParamName = 'bob'
    makeProper(properName)
    {
        addVocab(properName);
        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` and `vocabWords` properties in line with his self-introduction. Descriptions like `"You see {a bob} 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:

```
+ Fixture 'moustache; thin'
    ownerNamed = true
;

+ Wearable 'suit; smart business; clothes[pl]'
    wornBy = bob
;

+ stick: Thing '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' @highStreet
   "He's a tall man, wearing a smart business suit and sporting a thin
     moustache. "

   globalParamName = 'bob'
;

+ bobStanding: ActorState
   isInitState = true
   specialDesc = "{The subj bob} is standing in the street, looking in a shop
    window. "
   stateDesc = "He's looking in a shop window. "
;

+ bobWalking: ActorState
   specialDesc = "{The subj bob} 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 prepend actor to the method name (e.g. **actorBeforeAction(), actorBeforeTravel()** or **actorAfterAction()**) , otherwise we'll break the mechanism that farms these responses out to ActorStates in other cases:

```
mavis: Person 'Aunt Mavis; old frail; woman; her'    @lounge
   "Well past her prime, she is now looking distinctly frail. "
   actorAfterAction()
   {
      if(gActionIs(Yell))
         "Aunt Mavis glowers at you with a look fit to freeze the sun. ";
   }
;
```

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 subj bob} 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 subj bob} takes one look at you, and then turns away and
           starts walking briskly down the street. ";
         bob.setState(bobWalking);
      }
   }
;
```

Note the use of **setState(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.

The general rule is that where a method is used to define some behaviour specific to an ActorState, we can define the same method on the Actor, provided we prepend 'actor' to its name, and the actorXXX method will be used as well. But there are one or two exceptions, the most notable being **specialDesc** (defined on the ActorState) and **actorSpecialDesc** (defined on the Actor). The function of **specialDesc** is to provide a description of the Actor's presence in a room description. Normally, how an Actor should be described depends on what its current ActorState is, so the current ActorState's **specialDesc** will be used. It wouldn't make sense to use the **actorSpecialDesc** as well, since that would mean mentioning the same actor twice. However, if the Actor doesn't have a current ActorState (maybe because it's such a simple Actor that it doesn't need ActorStates), then the **actorSpecialDesc** will be used to supply the listing of the Actor in a room description.

Note that you *can* override **specialDesc**, **beforeAction()** and all the rest on the Actor object; it's just advisable not to do so, since if you do you'll break their library-defined behaviour and the ActorState mechanism won't work properly.

Actor defines a number of other methods which in one way or another rely on ActorStates, but one other of particular interest is **takeTurn()**. This 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 current 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.

14.4   Conversing with NPCs – Topic Entries

The standard form of conversation implemented in the adv3Lite 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 `ActorTopicEntry`. 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**

- `AltTopic` – provides an alternative response to any of the others when it's located in it.

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. There are also other kinds of TopicEntry, but the ones listed above are quite enough for now.

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), or it can be a Thing-derived class or a list of Things and/or classes. 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} 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> <.reveal bob-troubles>"

  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 adv3Lite 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.

But the best way to handle this case is with an **AltTopic**. To use an AltTopic, we locate it (with a + sign) in the TopicEntry for which it's an alternative response. It will then match whatever its parent TopicEntry matches, but will be used in preference to its parent when its isActive property is true. So, making use of an AltTopic, we'd probably handle the previous example 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 that we can have as many AltTopics as we like associated with any given TopicEntry; the one that will be used is the last one (i.e. the one furthest down in the source file) for which isActive is true.

Note also the **<.reveal bob-troubles>** in 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 adv3Lite defines a special kind of Topic Entry called a `DefaultTopic`. Or rather, adv3Lite 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 4; and the other four have a `matchScore` of 5. 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} 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} 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 four 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, 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).

4. TopicEntries can be located in a ConvNode, but that's something we'll come to later.

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 `scoreBoost` 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 `scoreBoost` 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 `scoreBoost` properties, Topic Entries in a TopicGroup behave just as if they were in that TopicGroup's container (actually a TopicGroup can do other things as well, but we'll get to them when we come to look at a special kind of TopicGroup called a ConvNode).

This may all become a little clearer with a skeletal example:

```
mary: Person 'Mary;; woman; her'
;

+ TopicGroup
    isActive = (mary.curState is in (maryWalking, maryTalking))
;

++ AskTopic @robert
   "blah blah"
;

++ TellTopic @tWedding
   "blah blah"
;

+ AskTopic @mary
  "blah blah"
;

+ maryWalking: ActorState
  specialDesc = "Mary is walking along beside you. "
;
```

```
++ AskTopic @tShopping
   "blah blah"
;

++ TellTopic @robert
  "blah blah"
;

++ TellTopic +10 @robert
  "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: ActorState
  specialDesc = "Mary is busily rehearsing an aria from <i>The Marriage of
   Figaro<i>. "
;

+ DefaultAnyTopic
   "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; one solution is to move the DefaultAnyTopic directly under Mary and then define `isActive = mary.curState == maryWalking` on it.

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 getting adv3Lite to suggest topics of conversation to the player. To do that, all we need to do is to define a `name` property on one or more TopicEntries 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 @mavis
  "<q>How are you today, Aunt Mavis?</q> you enquire.\b
   <q>Well enough,</q> she replies. "
  name = 'herself'
;

+ TellTopic @me
  "<q>You know aunt, I've been meaning to tell you about..."
  name = 'yourself'
;

+ GiveShowTopic  @ring
  "<q>That's a nice ring!</q> she declares. "
  name = (ring.theName)
;

+ AskForTopic @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 only displayed when they're reachable. Normally each TopicEntry will only be suggested once, that is only suggested until the player tries conversing about that topic for the first time. In fact, however, a TopicEntry 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 [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.

Normally the library will decide what to suggest a topic as. For example, if you give an AskTopic a name, it will be suggested as "You could ask Mavis about...", whereas if you give a TellTopic a name, it will be suggested as "You could tell Mavis about...". For some types of TopicEntry, such as AskTellTopic, there's more than one possibility, but then the library will always choose one of them (in the case of an AskTellTopic it will be "You could ask about..."). You can override the library's choice in such a case using the `suggestAs` property; for example:

```
+ AskTellTopic @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)
  suggestAs = TellTopic
;
```

This will make the above TopicEntry be suggested as "You could tell Merlin about the ring". Note that any value you give to a suggestAs property must be one the TopicEntry could match; it would have been pointless, for example, define **suggestAs = GiveTopic** on an **AskTellTopic.**

Conversely, in a situation like the one above where we want the name of the suggested topic to reflect that of the object it's matching, we could simply define **autoName = true** on the TopicEntry, and let the library work out the name for us; so the previous AskTellTopic could become:

```
+ AskTellTopic @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.
  autoName = true
  suggestAs = TellTopic
;
```

This would once again result in "You could tell Merlin about the ring", assuming this AskTellTopic was located in an Actor called Merlin.

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. Adv3Lite 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. These greeting protocols are based on a number of special Topic Entry classes used to say Hello and Goodbye:

- **HelloTopic** – A response to an explicit greeting; also used for an implicit greeting response if no **ImpHelloTopic** has been defined.

- **ActorHelloTopic** – The greeting when an NPC initiates the conversation (see below).

- **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).

The next issue is where these various kind of hello and goodbye topics should be located. For a very simple actor, one that makes little or no use of ActorStates, you could put them directly in the Actor, along with any other TopicEntries. More usually, however, you will want to put them in the actor's relevant ActorState, not least because the act of saying hello or goodbye may well make the Actor change state; how Bob should be described when he's talking to the player character is probably different from the way he should be described when he's about his own business, and this difference would normally be reflected by the use of different ActorStates.

The rule is that any greeting topic used at the start of a conversation (**HelloTopic** and its variants) should be located in the ActorState the Actor is in just prior to the conversation, while any greeting topic used at the end of the conversation (**ByeTopic** and its variants) should be located in the ActorState the Actor is in while the conversation is taking place; or to state the rule more succinctly, greeting topics should be located in the ActorState that will be active at the point at which they're invoked. So, for example, we might have:

```
bob: Actor 'Bob; tall thin; man; him' @street
   "He's a tall, thin man. "
;

+ bobLooking: ActorState
   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. '
  ]
  changeToState = bobTalking
;

+ bobTalking: ActorState
   attentionSpan = 5
   specialDesc = "Bob is standing by the shop window, waiting for you to
    speak. "
   stateDesc = "He's waiting for you to speak. "
;



++ ByeTopic
  "<q>Well, cheerio then!</q> you say.\b
   <q>'Bye for now,</q> Bob replies, turning back to the shop window.
    changeToState = bobLooking
;

++ BoredByeTopic
   "Bob gives up waiting for you to speak and turns back to the shop window. "
   changeToState = bobLooking
;

++ LeaveByeTopic
   "Bob watches you walk away, then turns back to the shop window. ";
   changeToState = bobLooking
;

++ 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. "
    changeToState = bobWalkingAway
;

++ AskTopic @bob
   "<q>How are you today?</q> you ask.\b
    <q>Fine, just fine,</q> he assures you. "
;
```

There a several additional points to note here. The first is the use of the
changeToState property on the various greeting topics. This tells the game to change
the Actor to the specified **ActorState** when the greeting topic is triggered, so that, for
example, any conversational command that triggers the **HelloTopic** while Bob is in
the **bobLooking** state will change Bob to the **bobTalking** state, while the **ByeTopic**,
BoredByeTopic and **LeaveByeTopic** will put Bob back into the **bobLooking** state. The
ActorByeTopic, however, would put him into a new **bobWalkingAway** state (not shown
here).

The `ByeTopic` would be triggered by an explicit BYE command. The `LeaveByeTopic` would be triggered by the player character walking away in the middle of the conversation. The `ActorByeTopic` would be triggered by Bob deciding to end the conversation for some reason of his own. To make the actor terminate the conversation in this way we can simply call `endConversation(reason)` on the actor object, e.g.:

```
bob.endConversation(endConvActor);
```

Here `endConvActor` means that the Actor has chosen to end the conversation. The other possible values for the *reason* parameter are `endConvBye` (the player character said goodbye), `endConvTravel` (the player character left the vicinity) and `endConvBoredom` (the Actor became tired of waiting for the player character to speak), but we seldom need to call `endConversation()` with any of these three other values of *reason*, since the library will do this for us as and when appropriate.

This leaves us with the `BoredByeTopic` (corresponding to `endConvBoredom`). This is triggered by the Actor having to wait too long for the player character to speak, or to put it a bit more precisely, by the number of turns since the player last issued a conversational command exceeding the actor's `attentionSpan`. The `attentionSpan` in question is that defined on the current ActorState, or, if there is none, on the Actor. The default value of nil means the Actor never gets tired of waiting; a numerical value would define the number of turns until the Actor gives up on the conversation.

14.7   Conversation Nodes

There come points in a conversation when a particular set of responses become appropriate that wasn'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 adv3Lite uses Conversation Nodes. These are objects of the `ConvNode` class. A `ConvNode` is like a `TopicGroup` in that we can put a number of Topic Entries in it, but it is used a little differently (although in adv3Lite `ConvNode` is in fact a subclass of `TopicGroup`).

At its simplest, the definition of a ConvNode object can be very simple indeed:

```
+ ConvNode
    convKeys = '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 (or other convKeys) 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
   "<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
  "<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, and we'll look at the complications shortly. But the next question to address is how we get the conversation into a particular ConvNode. We do so by outputting a special tag, either `<.convnode name>` or `<.convnodet name>`, where *name* is the name (technically the convKeys property) of the ConvNode we want to activate. The only difference between `<.convnode x>` and `<.convnodet x>` is that the latter is a convenient shorthand for `<.convnode x>` `<.topics>`; in other words it both activates the Conversation Node and then displays a list of suggested available topics.

So, for example, to activate the 'lighthouse' `ConvNode` shown above we'd typically do something 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.

This example shows how we use the `<.convnode>` tag by including it in a string of text to be displayed. This is how we always use it, but it does not follow that a `<.convnode>` tag can be used anywhere. This is partly because the library has to know which actor's Conversation Node is being invoked, and partly because the library has to do certain pieces of housekeeping behind the scenes to make the Conversation

Node mechanism work properly, and can only do that if the use of `<.convnode>` and `<.convnodet>` tags is restricted to predictable contexts. In particular, there are only four contexts in which they are guaranteed to work as expected:

- In the topicResponse() method of a TopicEntry (as shown above)

- In the eventList property of a TopicEntry (in one of the elements of the list)

- In the invokeItem() method of a ConvAgendaItem (which will be explained later in this chapter)

- In a string used as the argument to the `actorSay(str)` method of Actor (which method exists largely for this very purpose).

The last of these can be used, for example, to trigger a conversation in an `afterAction()` or `afterTravel()` method, for example:

```
bobStanding: ActorState
   specialDesc = "Bob is standing in the street, looking in a shop window. "
   afterTravel(traveler, connector)
   {
      inherited(traveler, connector);
      if(traveler == me)
         bob.actorSay('Bob turns to greet you. <q>Hello, there!</q>
            he says. <q>I\'ve been thinking - would you like to
            come and see the lighthouse?</q> <.convnode lighthouse>
         <.state bobTalkingState>');
   }
;
```

This, incidentally, illustrates the use of another tag, the `<.state x>` tag which can be used to change an actor's ActorState to x; precisely the same limitations on its use apply as to the `<.convnode x>` tag.

As we've defined this ConvNode so far, it will last only until the player makes some conversational response. 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 a bit more work. In particular, 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 NPC's current ActorState, plus also perhaps any defined directly in the Actor, will also be available). The easiest way to handle this is to define a single DefaultAnyTopic (also located within the ConvNode) to field all the conversational responses other than the ones we have specifically catered for, perhaps combining it with a ShuffledEventList to vary the responses:

```
++ 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>` tag here. This is used to tell the system *not* to leave the current ConvNode when the item in which it occurs is triggered. We thus need to include it in each of the responses in the DefaultAnyTopic to ensure that all of them keep the current ConvNode active.

Note that there's no law that says that a ConvNode *has* to have a DefaultAnyTopic, or that if it does it always has to use a `<.convstay>` tag. For example, if an Actor asks a question that might be taken as a rhetorical one, we could have it set up a ConvNode to field a YES or NO response if the player chooses to give one, but which otherwise lets the moment pass if the player prefers to talk about something else. Again, even if we do include a DefaultAnyTopic, it doesn't have to use a `<.convstay>` tag; it could instead have the NPC complain about the player character's attempt to change the subject and then move the conversation on. We do, however, need to use the `DefaultTopic/<.convstay>` mechanism if we want the ConvNode to remain active until the player chooses one of the TopicEntries we've explicitly defined for that ConvNode.

In such a case, we may well want our NPC to keep insisting on an answer if the player insists on entering a series of non conversational commands (such as I, LOOK, X BOB, THROW BALL AT VASE) while the NPC is waiting for an answer. To do that we can define a `NodeContinuationTopic`, which is again located within the ConvNode it's designed to continue, for example:

```
 ++ NodeContinuationTopic: 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
   }
;
```

In this example we used a `ShuffledEventList` to vary Bob's 'nag' message, and defined `eventPercent = 67` so that the 'nag' message would only appear on average on two turns out of three.

There's one more aspect we may want to control if we're defining a ConvNode where our NPC is insisting on an answer. In principle the player could frustrate the NPC's insistence by ending the conversation, either by saying BYE or by walking away or by waiting until the NPC exhausted its `attentionSpan`. To control what happens in such cases we can define a `NodeEndCheck` object to go in our ConvNode, and define its `canEndConversation()` method to determine what should happen in such cases:

```
++ NodeEndCheck
   canEndConversation(reason)
   {
      switch(reason)
      {
         case 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;

         case 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;
      }
   }
;
```

The `canEndConversation(reason)` method should return true to allow the conversation to end, or nil to prevent it. Returning `blockEndConv`, as do two of the branches of the switch statement in the above example, also prevents the conversation from ending but additionally tells the game that the actor has conversed on this turn. This prevents the game from trying to make the actor carry out any additional conversation on the same turn, such as displaying one of the items from the `NodeContinuationTopic`, which would look superfluous immediately following Bob's responses to attempts by the player character to say goodbye or walk away.

Incidentally, you can also define `canEndConversation(reason)` on an ActorState to control whether conversation can end while the actor is in that state, or `actorCanEndConversation(reason)` on the Actor object to determine if the Actor will allow the conversation to end independently of state.

One last point: we've emphasized the use of the `<.convstay>` tag to make sure that a conversation remains at a particular node when we don't want to leave it, but it can, of course, be equally useful to use a `<.convnode>` or `<.convnodet>` in one ConvNode's TopicEntries to change to a different ConvNode, perhaps thereby setting up a threaded conversation that passes through a number of different ConvNodes, each of which offers the player a number of options relevant to the particular point in the conversation that has been reached.

14.8   Special Topics – Extending the Conversational Range

In the Conversation Node example above we gave the player the option of replying YES or NO; anything else was treated as an attempt to change the subject. Within the ask/tell system we have seen so far we could have implemented other options, such as ASK BOB ABOUT THE TROUBLES or TELL BOB ABOUT YOUR PHOBIA or even ASK BOB FOR ADVICE, but there's a limit to the expressiveness of such conversational commands. For example, when we ASK BOB ABOUT THE TROUBLES, so we want to know when they started, or how long they lasted, or why they were so terrible, or what they were about? Of course, *not* allowing the player to specify which is meant can have its advantages, since the game author can then control how the question is meant and how it should be answered, making it easier to script the conversation. But sometimes we may want to give the player some more expressive options, and to do that we can make use of SpecialTopics.

Or rather, we can make use of one of the two subclasses of SpecialTopic defined in the adv3Lite library: **SayTopic** and **QueryTopic** (we don't use SpecialTopic directly). The first of these allows the player to say almost anything we like (within reasonable length); the second extends the range of questions that can be asked to ASK WHO/WHAT/WHY/WHERE/WHEN/HOW/IF such-and-such.

We can define a **SayTopic** in one of two ways. The first is to define a Topic object (or use one we've already defined) to give the vocab for what the player character can say. For example, to let the player character say he's afraid to go, you could define:

```
tAfraidToGo: Topic 'you\'re afraid to go; you are i\'m i am';
```

The corresponding SayTopic can then be defined simply as:

```
+ SayTopic tAfraidToGo
    "<q>I'm afraid to go to the lighthouse, after what I've heard,</q>
     you confess.\b
     <q>You shouldn't believe all those scare stories and old wives' tales,</q>
     Bob chides you. "
;
```

This will then respond to **say you're afraid**. Because of the way we've defined the vocab property of tAfraidToGo it will also respond to most of the obvious variants, such as **say you are afraid**, **say I'm afraid,** and **say I am afraid**. In fact, it will even work if the player omits the say and just types **i'm afraid**.

We may think that the way the tAfraidToGo's vocab property is defined here is a bit too liberal here, since the SayTopic could even respond to just **are** or **am**. This might be a case where it would be helpful to mark such auxiliaries as weak tokens, so that the word 'afraid' has to appear in the player's input for the SayTopic to be triggered:

```
tAfraidToGo: Topic '(you\'re) afraid (to) (go); (you) (are) (i\'m) (i) (am)';
```

This general approach is fine if we're going to use the tAfraidToGo Topic elsewhere in our game, but if we're having to define it just for the purposes of this one SayTopic then it may seem a bit laborious to have to go through the additional step of doing so. Adv3Lite therefore lets us define such a Topic along with the SayTopic all in one go, like this:

```
+ SayTopic '(you\'re) afraid (to) (go); (you) (are) (i\'m) (i) (am)'
    "<q>I'm afraid to go to the lighthouse, after what I've heard,</q>
     you confess.\b
     <q>You shouldn't believe all those scare stories and old wives' tales,</q>
     Bob chides you. "
;
```

By default a SayTopic will always be suggested (since the player can hardly be expected to guess the syntax), using the name property of its associated Topic as its own **name** property (in other words, a SayTopic has **autoName = true** by default). By default the form of the suggestion will be "You could say [topic name]". In some cases, though, you may want to exclude the 'say' from the name of the suggestion, which you can do by setting **includeSayInName** to nil. For example:

```
+ SayTopic 'be evasive'
    "<q>You mumble something incoherent about keeping confidences...</q>"
    includeSayInName = nil
;
```

Such a topic would be suggested in the form "You could be evasive".

There are some restrictions on what kind of text can match a SayTopic without an explicit SAY however. These arise from the fact that the parser will attempt to match player input to an ordinary command before it tries it as an attempt to say something. Thus if the player types **take me** (for example), it will never be interpreted as **say take me**. Usually this isn't too much of a restriction in practice, but we may run up against it occasionally.

For example, suppose we wanted one possible response in a ConvNode to be **tell the truth**. We can't define this as a SayTopic with vocab 'tell the truth' and an autoName of nil, since the input **tell the truth** will be parsed as a TELL command before it's ever tried as a SAY command. In this case we should need to define a Topic matching 'truth' and then use a TellTopic to field the response:

```
+ TellTopic @tTruth
    "<q>Well, to tell the truth,</q> you begin... "
    autoName = true
;

tTruth: Topic 'truth'
```

And so on for any other conversational commands beginning **tell** such as **tell a white lie**.

A `QueryTopic` is like a SayTopic, except that it's used to ask a question. Just as with a SayTopic the question asked can either be defined on a separate Topic object, or defined in-line with the QueryTopic. In addition, however, we have to define what type of question we're asking: who, what, why, how, when, if or whether. This is defined on a separate property call `qType`, which makes it easier for the parser to recognize and match such questions, although we'd normally define this property implicitly through the QueryTopic template. So, for example, if we want to be able to ask Bob where the lighthouse is, we could do it like this:

```
+ QueryTopic 'where' @tLighthouseIs
    "<q>Where is the lighthouse, anyway?</q> you ask.\b
     <q>On the promontory, overlooking the cove,</q> he replies. "
;

tLighthouseIs: Topic 'lighthouse[n] (is)'
;
```

Or all in one like this:

```
+ QueryTopic 'where' 'lighthouse[n] (is)'
    "<q>Where is the lighthouse, anyway?</q> you ask.\b
     <q>On the promontory, overlooking the cove,</q> he replies. "
;
```

Or even, if we prefer, by merging the qType into the topic's vocab property (although the library will then separate them out again for us):

```
+ QueryTopic 'where lighthouse[n] (is)'
    "<q>Where is the lighthouse, anyway?</q> you ask.\b
     <q>On the promontory, overlooking the cove,</q> he replies. "
;
```

All three of these forms will respond to **ask bob where the lighthouse is** or **ask where the lighthouse is** or even just **where is the lighthouse**. Since the player may use the third form you'll often need to provide extra vocab words to cater for it, for example:

```
+ QueryTopic 'where' 'you are; am I'
   "<q>Where am I?</q> you ask..."
;
```

This will respond to either **ask where you are** or **where am I** (along with a number of other variants).

Like SayTopic, QueryTopic has an autoName of true by default, so you'll generally want to arrange things such that the name part of the associated Topic's vocab property makes sense in a suggestion like "You could ask where [topic name]".

Finally, you can make a QueryTopic match more than one qType word by separating the alternatives with a vertical bar; this is most useful with 'if' and 'whether', for example:

```
+ QueryTopic 'if|whether' 'he saw the troubles for himself; you see'
    "<q>Did you see...</q> "
;
```

We add 'you see' to the vocab here, since this QueryTopic could then indeed match **did you see the troubles**.

Although we began this section by talking about possible responses in a Conversation Node, SpecialTopics and QueryTopics are by no means restricted to Conversation Nodes; they can be used anywhere any other conversational TopicEntry can be used.

14.9 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 seem more realistic if we can make them pursue their own agendas. We can do this with the `AgendaItem` class. By 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

    invokeItem()
    {
      "Bob wanders off down the street and disappears round a corner. ";
      getActor.moveInto(nil);
    }
;

+ 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 (which, in this case, is at once), whereupon the AgendaItem will be removed from Bob's `agendaList`. In this example the invokeItem() method makes Bob walk away (and then leave the game map). 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 Mavis should happen upon Bob while bobWanderAgenda is about to send Bob on his walk, Bob will simply walk away 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. This is fine if the player character can see what Bob is doing, but might be inappropriate if the player character is in a completely different location when the AgendaItem is invoked. If we want an AgendaItem to display some text only if the player character can see its actor, we can use the **report()** method (note, this is a method of AgendaItem, not a generally available function), for example:

```
+ bobWanderAgenda: AgendaItem
    isReady = (bob.curState == bobWalking)
    initiallyActive = true
    agendaOrder = 10

    invokeItem()
    {
      report('Bob wanders off down the street and disappears round a
       corner. ');
      getActor.moveInto(nil);
    }
;
```

To summarize so far: 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*.

14.10   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 (preferably via the **actorSay()** method of the NPC in question). Or we can use actorSay() in the **afterAction(), beforeAction(), afterTravel()** or **beforeTravel()** method of an ActorState (remembering to call it on the Actor). 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. We can basically just display some text, particularly 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
    }
;
```

If, in addition, we want the actor to change state and/or enter a Conversation Node we can use a **<.state newState>** tag and/or a **<.convnode node-name>** or **<.convnodet node-name>** tag (the last of these also displays a list of suggested topics). As an alternative to using a **<.state newState>** tag we could call **getActor.setState(newState)**. For example:

```
+ bobLighthouseAgenda: ConvAgendaItem
    isReady = (inherited && bob.canSee(lighthouse))
    invokeItem()
    {
       "<q>Look! There's the lighthouse!</q> Bob declares. <q>Shall we go
        in?</q> <.convnode enter-lighthouse> <.state bobByLighthouseState>";
       isDone = true
    }
;
```

There are three slightly different situations in which a `ConvAgendaItem` might be triggered:

1. No conversation is currently in progress and the NPC is starting one.

2. A conversation is in progress and the NPC is taking advantage of a lull in the conversation (i.e. a turn on which the player does not enter a conversational command) to pursue its own conversational agenda.

3. The player enters a conversational command for which there's no specific response and the response is handled by a `DefaultAgendaTopic`.

We may wish to vary what the NPC says according to which of these situations obtains when the `ConvAgendaItem` is triggered. We can do this by checking the value of its `reasonInvoked` property, which will be one of `InitiateConversationReason` (the NPC is starting a new conversation from scratch), `ConversationLullReason` (the NPC is taking an advantage of a lull in the conversation) or `DefaultTopicReason` (the NPC is using a default response to change the subject). So, for example, the previous `ConvAgendaItem` could become:

```
+ bobLighthouseAgenda: ConvAgendaItem
    isReady = (inherited && bob.canSee(lighthouse))
    invokeItem()
    {
        if(reasonInvoked == DefaultTopicReason)
            "<q>Never mind that now, that's the lighthouse just over there.</q>
            Bob interrupts. ";
        else
            "<q>Look! There's the lighthouse!</q> Bob declares. ";
    "<q>Shall we go in?</q> <.convnode enter-lighthouse>
        <.state bobByLighthouseState>";
        isDone = true
    }
;
```

A `DefaultAgendaTopic` can be used in place of the normal kinds of DefaultTopic to allow the NPC to seize the conversational agenda when the player enters a conversational command that's not dealt with by some more specific kind of TopicEntry. This may often be better than a bland default reply, and can be used to create a more pushy NPC determined to pursue his or her own conversational agenda by changing the subject as soon at the player tries to talk about something the NPC isn't immediately interested in. A `DefaultAgendaTopic` is only active if it has something in its `agendaList`, so although you'd use it like any other kind of DefaultTopic, you'd also want to define one or more other kinds of DefaultTopic to field the conversational commands the `DefaultAgendaTopic` will ignore when it has no AgendaItems to trigger. It's also conceivable that a DefaultAgendaTopic will have items in its agendaList that are not due to be executed, so it's a good idea to define a default response on a `DefaultAgendaTopic` to be used in such a situation. Typically,

then, you might define a `DefaultAgendaTopic` like this:

```
+ DefaultAgendaTopic
    "Bob merely grunts in reply. "
;
```

Note that we referred to the `DefaultAgendaItem`'s agendaList. This is kept separately from the Actor's agendaList (the list of AgendaItems waiting to be executed), since there may be items in the latter that aren't suitable in the former; in particular it will usually make sense for a `DefaultAgendaItem` to trigger only `ConvAgendaItems` and not ordinary `AgendaItems`. The Actor's `addToAgenda(item)` method adds item only to the Actor's agenda. To add item to the Actor's ConvAgendaItem's agenda as well, use either `addToAllAgendas(item)` (called on the Actor) or the tag `<.agenda item>` (which can be used in contexts where a `<.convnode>` tag would be legal).

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 who comments on some of the locations as she visits them for the first time. We could implement this as follows:

```
+ sallyFollowing: ActorState
    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.

14.11   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 adv3Lite will respond to all such commands with "Bob has better things to do" (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 what an NPC will do with a command is `handleCommand(action)`, where *action* is the action that the player wants the NPC to perform. Normally, however, this should just be left to get on with its job, which is to rule out system actions (it doesn't make sense to issue a command like **bob, undo**), and translate various others (e.g. **bob, give me the cake** is turned into **ask bob for cake**). If it's not one of the actions that need translating (and most don't), then `handleCommand()` will attempt to find a `CommandTopic` to handle it, so it is through the use of `CommandTopics` that we mostly determine how an NPC will respond to orders. If we don't define any `CommandTopics` at all, or there's no `DefaultCommandTopic` to handle the particular command that's just been issued, then the fall-back position is to display the actor's `refuseCommandMsg`, which by default simply says something along the lines of "Bob has better things to do."

We could, of course, use a different message here if we wanted to, and this would be the simplest way to customize the way in which an actor responds to commands; for example:

```
mavis: Actor 'Aunt Mavis; old; woman; her'
   refuseCommandMsg = '<q>Don\'t you tell me what to do, young man!</q>
      she snaps. '

;
```

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 calling the method `nestedActorAction()` in the topicResponse():

```
+ CommandTopic @JumpAction
   topicResponse()
   {
      "<q>Jump!</q> you cry.\b";
      nestedActorAction(bob, Jump);
   }
```

```
;
```

But where we just want the actor to obey the command as issued, it's easier just to set the CommandTopic's `allowAction` property to true:

```
+ CommandTopic @JumpAction
   "<q>Jump!</q> you cry."
    allowAction = true
;
```

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. "
;
```

Here, though, we may want to include the player's command as well as the NPC's response, since otherwise the interchange may look a little odd, especially if it's the opening interchange in a conversation. We can do this by using the `actionPhrase` method to return a string representing the command just issued, in the form "jump" or "take the red ball"; for example:

```
+ DefaultCommandTopic
   "<q>Aunt, I think you should <<actionPhrase>>,</q> you suggest.\b
    <q>Don't you tell me what to do, young man!</q> she snaps. "
;
```

In many situations we won't just want to match a CommandTopic on the kind of action (e.g. Jump, Take, or PutIn), but, in the case of a transitive action, on the particular objects involved in that action. For example, we'd probably want to handle **mavis, eat cake** differently from **mavis, eat table** or **mavis, eat your hat**, and distinguish **bob, put the red ball in the brown box** from **bob, put the flaming torch in the vat of petrol**. We can achieve this by defining the `matchDobj` and `matchIobj` properties on a `CommandTopic`, for example:

```
+ CommandTopic @PutInAction
   "<q>Put the red ball in the brown box, would you?</q>\b
    <q>Okay,</q> Bob agrees. "
    matchDobj = redBall
    matchIobj = brownBox
    allowAction = true
;
```

14.12 NPC Travel

There may be some NPCs who have a good reason for staying right where they are throughout the course of a game, but the chances are that at least some of our NPC will need to move around. The simplest way to move an NPC from place to place is to call its **moveInto(dest)** method to move it to *dest*, but sometimes we may want something a bit more sophisticated than instant teleportation.

Rather than simply moving an NPC round the game map by programmatic fiat, for example, we may want to move it via a particular TravelConnector, the better to simulate how actors actually move round the map, and to ensure both that any side-effects of the travel are carried out and that any travel barriers are given due opportunity to veto the travel. We can do that by calling **travelVia(npc)** on the *travel connector* we want the npc to traverse. If the player character is in a position to see the travel we can also call **sayDeparting(conn)** on the actor to report the travel. It's usually more convenient, however, to combine these steps into a single step by calling **travelVia(conn)** on the *Actor.* This first calls **sayDeparting()** on the Actor if the player character can see the actor departing; it then carries out the travel via *conn*; it finally calls **sayArriving(oldLoc)** on the Actor provided the player character can see the arrival but did not see the departure (where *oldLoc* is the location the Actor has just arrived from). This in turn calls **sayArriving(oldLoc)** on the current ActorState if there is one, or **actorSayArriving(oldLoc)** if there isn't. By default this just displays a message like "Bob arrives in the area", so game code may often want to override this to something more specific. Alternatively you can suppress the arrival message altogether by calling **travelVia(conn, nil)**, with nil as the optional second parameter.

Sometimes we may want an NPC to follow the player character around for a while. To do that we can simply call the NPC's **startFollowing()** method, and the NPC will attempt to keep following the player character around until we call **stopFollowing()**. Each turn when the NPC is following the player character the **sayFollowing(fromLoc)** method is called on the NPC's current ActorState (if there is one), or else **sayActorFollowing(fromLoc)** is called on the NPC, where *fromLoc* is the location the NPC is following from. By default these just display something along the lines of "Bob follows behind you", but obviously they could be overridden to say something more specific to the character or the game.

Conversely, we may want to let the player character follow an NPC around. To do that we create a **FollowAgendaItem** for the NPC which defines where it will go when it's followed. In fact, a **FollowAgendaItem** is a bit of a hybrid between an AgendaItem and an ActorState, in that it fulfils some of the functions of each. The properties and methods we'd typically need to define on a **FollowAgendaItem** are:

- **connectorList** – a list of TravelConnectors (which may just be rooms, but could include Doors, Stairways and the like) through which the NPC wants to

lead the player character.

- **specialDesc** – this is used in place of the regular specialDesc to list the NPC in a room description, and should normally say that the NPC is waiting for the player character to follow him or her in a particular direction; the default implementation already does this in a plain vanilla way, but game code may want to override this to something more specific to the game.

- **arrivingDesc** – the specialDesc to be shown on the turn on which the player character follows our NPC to a new location. By default we just show our specialDesc (as defined immediately above).

- **noteArrival()** - this method is called when the NPC arrives at his or her destination (i.e., when the NPC has traveled via the last connector in the connectorList). By default this method does nothing but it could be overridden, for example, to put the NPC into a new ActorState.

- **sayDeparting(conn)** – the message to display to describe our NPC's departure from its current location via conn. The default behaviour is to call **conn.sayActorFollowing()** to describe the player character following the NPC via conn.

- **beforeTravel(traveler, connector)** – this notification is called just before *traveler* attempts to depart via *connector*. By default it does nothing, but it could typically be used to veto an attempt by the player character to travel anywhere other than where the NPC is trying to lead him or her. We'd normally do this by displaying some text representing the NPC's protest and then using **exit** to prevent the travel.

- **cancel()** - we can call this method on the FollowAgendaItem to cancel it before our NPC reaches its destination.

- **nextConnector** – the TravelConnector through which the NPC wishes to lead the player character next (this should be treated as read-only).

- **nextDirection** – the direction in which the NPC wishes to lead the player character next (this should also be treated as read-only, and returns a direction object, e.g. northDir).

For example, suppose Bob wants to lead the player character out through the front door, along the main road, across the field, down the cliff path and into the lighthouse; we might define a **FollowAgendaItem** for him thus:

```
+ bobLeading: FollowAgendaItem
    connectorList = [frontDoor, mainRoadSouth, largeField, cliffPath,
        lighthouseDoor]
    beforeTravel(traveler, connector)
    {
      if(traveler == gPlayerChar && connector != nextConnector)
      {
        "<q>No, <i>this</i> way,</q> Bob insists, pointing firmly to the
```

```
            <<nextDirection.name>> ";
            exit;
        }
    }

    noteArrival()
    {
        getActor.setState(bobInLighthouseState);
    }
;
```

Note that we would have to call `bob.addToAgenda(bobLeading)` for this to have any effect, and that we'd normally do so just at the point at which Bob had indicated that the player character should follow him. The player could then make the player character follow Bob either by issuing a **follow bob** command, or by issuing a travel command that would take the player character in the direction Bob wants to lead him.

All the ways of moving NPCs we've looked at so far presuppose that we know exactly which connectors we want to move the NPC through. This may often be the case, but suppose we want to send an NPC to a particular destination when we don't know in general where he or she will be starting out from? In this case we can use the `findPath` method of the `routeFinder` object to calculate the route for us. For example, suppose we want to send Bob to the lighthouse from wherever he happens to be; then we could obtain the route thus:

```
    bobRoute = routeFinder.findPath(bob.getOutermostRoom, lighthouse);
```

The route that's returned will either be nil (if there's no path available to the lighthouse) or a list of two-element lists, the first element of which will be the direction to travel in and the second the destination it leads to, for example:

```
[[eastDir, mainRoad], [southDir, mainRoadSouth], [southWestDir, largeField],
[westDir, cliffPath], [westDir, lighthouse]]
```

To use this route we might want to convert it into a list of TravelConnectors to be traversed, which we could do with code like the following:

```
local bobRoute = routeFinder.findPath(bob.getOutermostRoom, lighthouse);
local loc = bob.getOutermostRoom;
local path = [];
foreach(item in bobRoute)
{
   path += loc.(item[1].dirProp);
   loc = item[2];
}
```

Then, for example, we could set the value of the `connectorList` of a `FollowAgendaItem` to `path`, or else write an ordinary AgendaItem to send Bob one step along path each turn (by calling `bob.travelVia(path[idx])`, where idx was a counter of steps taken).

14.13 Afterword

Writing life-like NPCs is usually the most complex and challenging task an IF author can face. This chapter has been correspondingly long and complex; there's a lot of material to take in here, and we haven't covered all there is, especially on the conversation side. We have, however, covered enough (and probably more than enough) to illustrate the basics, and if you can master the contents of this chapter, you'll be well on the way to writing well-implemented NPCs in your own work. For further information about the nooks and crannies not covered here you can turn to the *adv3Lite Manual* if and when you need it.

You may, for example, have noticed that a `convKeys` property has been mentioned more than once in passing without ever really being explained. It can be used for a number of purposes. We've seen part of its use in relation to ConvNodes, but it could also be used (for example) to make the same TopicEntry appear in more than one ConvNode, or to set up a TopicEntry that suggests other topics for discussion, so that, for example, the player could type **talk about the troubles** and be rewarded with a list of questions s/he could ask or statements s/he could make about the troubles. This, incidentally, reveals a further type of TopicEntry we've not yet mentioned, the `TalkTopic` and its various relatives (at its simplest a TalkTopic can be used much like an AskTopic, except that it responds to **talk about x** rather than **ask about x**). We can also use various tags not yet mentioned like `<.activate key>` and `<.suggest key>` or `<.arouse key>` to control what TopicEntries should be made active or suggested, where *key* relates to the convKeys property of one or more TopicEntries. At this point we merely mention these possibilities without explaining them so that you're aware of their existence; you can read the Actors section of the *Library Manual* if and when you want to know more about them.

One additional tag it might be worth knowing about at this stage, however, is the `<.inform info>` tag. This works just like the `<.reveal info>` tag described back in Chapter 7, except that instead of recording what the player character knows, such as what an NPC has just revealed to the player character in conversation, it records what the NPC knows, in particular what the player character has just informed the NPC about in conversation. This can then be used to track what different NPCs have been told. To test whether an NPC has been informed about something you can use the macro `gInformed('info')`. Note that for this to work it can only be used in a context where the library can figure out which actor's knowledge you're asking about, which will be on any object that defines the `getActor` method (i.e. an ActorState, TopicGroup, ConvNode, AgendaItem, TopicEntry or, indeed, Actor). While 'info' can be any arbitrary single-quoted string you like, it's a good idea to use the same string to mean the same thing in both `<.reveal info>` and `<.inform info>` tags.

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.

15 MultiLocs and Scenes

15.1 MultiLocs

At a stretch we could regard MultiLocs and Scenes as complementary: MultiLocs cater for one object being in several places at the same time, while Scenes divide up time rather than space. In reality, though, the only reason for putting them both in the same chapter is that we need to cover them both but neither merits a whole chapter to itself. We'll start with MultiLocs.

Generally, the whereabouts of an object in an adv3Lite 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 a Decoration (a bunch of trees in a forest, say, or a plain vanilla ceiling in a house) identical instances of which occur in several places.

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]`. Note that this property (or `initialLocationList`, which we can use for the same purpose) can contain Regions as well as Rooms and Things.

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` (supposing we had defined a suitable OutdoorRoom class). 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 also define an **exceptions** list so that the MultiLoc will start out in every location defined by either of the first two methods except for those listed in **exceptions**.

So, for example, if we'd defined OutdoorRoom and ForestRoom classes, where ForestRoom was a subclass of OutdoorRoom, we might define a Distant object to represent the sun in all those places like so:

```
sun: MultiLoc, Distant 'sun; bright'
   "It's too bright to look at for long. "
   initialLocationClass = OutdoorRoom
   isInitiallyIn(obj) { return !obj.ofKind(ForestRoom); }
;
```

Similarly, if we wanted a fountain to stand at the centre of a square comprising four different rooms, we could define it thus:

```
fountain: MultiLoc, Container, Fixture 'fountain; ornamental'
   "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). It's also legal to use one or more Regions.

These two uses of MultiLoc are quite safe, because both the sun and the fountain are the same sun or fountain from whichever location they're viewed. If we're going to use MultiLoc to represent qualitatively similar but numerically distinct items like trees in a forest or ceilings in the room of a house we have to be a bit more careful. Normally we should only implement simple Decoration objects in this way, that is, objects that allow only the very minimum of interaction (examining, and perhaps smelling, listening to or feeling). We certainly shouldn't use MultiLocs to represent any set of similar objects where one of those objects might change state independently of the others. Don't, for example, use a MultiLoc, Decoration to represent trees in the forest and then allow the player to start cutting down trees, since once one tree is cut down, they all will be! In such a case it's better to create a custom Tree class and arrange for one instance of it to be in each forest location.

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 MultiLoc to give

sensible results). To a limited extent we can also determine the location of a MultiLoc by inspecting its `location` property. Strictly speaking this should be meaningless, because a MultiLoc is normally in several locations at once, but for many purposes what we're actually interested in is whether the MultiLoc is in the same location as the actor, or at least a location in the same room as the actor, so MultiLoc's location property will return nil if the MultiLoc isn't anywhere at all, a location within the current actor's current room (or the actor's current room itself) if the Multiloc is present there, or the location the MultiLoc was last seen at, if there is one, or, failing all that, the first location in the MultiLoc's `locationList` (which may have been updated since it was originally defined). This should generally give a usable result, provided that it's used with reasonable awareness of its limitations.

We can use `moveInto(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).

15.2   Scenes

Rooms and Regions can be used to divide up a game spatially. If we want to divide it up temporally we can use Scenes. A `Scene` is simply an object that represents a state of affairs that is ongoing for the time being, but which may have a definite start-point and end-point within the game. (Users familiar with Inform 7 might like to know that adv3Lite Scenes are very similar to Inform 7 Scenes, from which they are shamelessly 'borrowed').

To expand on that slightly, we can define a `Scene` such that it starts when a particular condition becomes true, and ends when another condition becomes true. We can also define what happens when the Scene starts and ends, and what happens each turn when it is current. We can test how long a Scene has been going on for, whether it's currently happening or whether it has happened and how it ended, and we can use the currency of a particular Scene on the during condition of a Doer (as explained in Chapter 6 above).

The properties and methods we can define on Scene are as follows:

- `startsWhen`: an expression or method that evaluates to true when you want the scene to start.

- **endsWhen**: an expression or method that evaluates to something other than nil when you want the scene to end. Often you would simply make this return true when you want the scene to end, but if you wanted to note different kinds of scene ending you could return some other value (which could be a number, a single-quoted string, an enum or an object) to represent how the scene ends.
- **isRecurring**: Normally a scene will only occur once. Set **isRecurring** to true if you want the scene to start again every time its startsWhen condition is true.
- **whenStarting**: A method that executes when the scene starts; you can use it to define what happens at the start of the scene.
- **whenEnding**: A method that executes when the scene ends; you can use it to define what happens at the end of the scene.
- **eachTurn**: A method that executes every turn that the scene is active.

In addition your code can query the following properties of a Scene object (which should be treated as read-only by game-code since they're updated by library code):

- **isHappening**: Flag (true or nil) to indicate whether this scene is currently taking place.
- **hasHappened**: Flag (true or nil) to indicate whether this scene has ever happened (and ended).
- **startedAt**: The turn number at which this scene started (or nil if this scene is yet to happen).
- **endedAt**: The turn number at which this scene ended (or nil if this scene is yet to end).
- **timesHappened**: The number of times this scene has happened.
- **howEnded**: An optional author-defined flag indicating how the scene ended (this could be a number, a single-quoted string, an enum or an object).

The **howEnded** property deserves a further word of explanation. You can use this more or less how you like, but one coding pattern might be to use custom objects to represent different endings and then make use of the methods and properties of your custom objects. For example, suppose a certain scene ends tragically if Martha is dead but happily if you give Martha the gold ring, you might do something like this:

```
class SceneEnding: object
   whenEnding() { }
;

happyEnding: SceneEnding
   whenEnding() { "A surge of happiness washes over you..."; }
;

tragicEnding: SceneEnding
   whenEnding() { "You feel inconsolable at your loss..."; }
;

marthaScene: Scene
   startsWhen = Q.canSee(me, martha)

   endsWhen()
```

```
    {
        if(martha.isDead)
           return tragicEnding;

        if(goldRing.isIn(martha))
           return happyEnding;

        return nil;
    }

    whenEnding()
    {
       if(howEnded != nil)
          howEnded.whenEnding();
    }
;
```

One special point to note: if you define the **startsWhen** condition of a Scene so that it is true right at the start of play (e.g. **startsWhen = true**), the Scene will indeed start out active, but its **whenStarting()** method will execute just *before* the first room description is displayed. If you want it to display something just after, you could use its **whenStarting** method to set a **OneTimePromptDaemon** for the purpose, e.g.:

```
 introScene: Scene
    startsWhen = (harry.isIn(harrysBed))
    endsWhen = (!harry.isIn(harrysBed))


    whenEnding()
    {
        "Harry stretches and yawns, before staggering uncertainly across the
        room. ";
    }

    whenStarting()
    {
        new OneTimePromptDaemon(self, &groan);
    }

    groan = "Harry groans. "
;
```

16 Senses and Sensory Connections

16.1 The Four Other Senses

Adv3Lite comes with handling for five senses: sight, sound, smell, touch and taste. The most elaborate provision is made for sight; the provision for the other four senses is fairly basic, but slightly fuller for sound and smell than for taste and touch.

We *can* handle all four of the non-visual senses in a similar fashion by defining `listenDesc`, `smellDesc`, `feelDesc` and `tasteDesc` to provide responses to **listen to**, **smell**, **feel** and **taste** respectively; for example:

```
apple: Food  'apple; round green red sweet firm juicy; fruit[pl]'
   "It's round, with patches of both red and green. "
   feelDesc = "It feels reassuringly firm. "
   tasteDesc = "It tastes sweet at juicy. "
   listenDesc = "The apple is obstinately silent. "
   smellDesc = "There's the faintest sweet apple smell. "
;
```

In order to be able to taste something the player character has to be able to touch it, and in order to be able to touch it the player character has to be in the same room as the object s/he wants to touch without any obstacles (such as a closed container, or a `checkReach()` method that forbids reaching) to prevent the object from being touched. As we shall see below, it's sometimes possible to see, smell and hear objects that are in remote locations, but it is never possible to touch or taste them.

Smell and sound are different from taste and touch in another respect. The commands **touch** and **taste** can't be used without specifying an object to be touched or tasted, e.g. **touch apple** or **taste apple**. But **smell** and **listen** can both be used intransitively, without specifying any particular object, in which case they'll list the `smellDesc` or `listenDesc` of every object in scope that defines a `smellDesc` or `listenDesc`. If we want to exclude an object from such a listing (because we reckon its sound or smell wouldn't obtrude on the player character unless s/he explicitly listened to or smelt that particular object), we could set its isProminentNoise and/or isProminentSmell properties to nil, e.g.:

```
apple: Food  'apple; round green red sweet firm juicy; fruit[pl]'
   "It's round, with patches of both red and green. "
   feelDesc = "It feels reassuringly firm. "
   tasteDesc = "It tastes sweet at juicy. "
   listenDesc = "The apple is obstinately silent. "
   smellDesc = "There's the faintest sweet apple smell. "
   isProminentNoise = nil
   isProminentSmell = nil
;
```

One further respect in which sound and smell differ from the other three senses is

that we can define objects to represent a `Noise` or a `Odor` distinct from any physical object that may be giving off that noise or smell. For this we just define the `desc` property of the Noise or Smell to respond to **listen to noise** or **smell odor** (or to examining either of them). For example:

```
cave: Room 'Large Cave'
   "There's general smell of dampness here, as well as the sound of
     dripping water. "
;
+ Odor 'smell of dampness; pervasive musty; damp'
   "It's a pervasive, musty smell. "
;
+ Noise 'sound of dripping water; continuous'
    "You can't locate where it's coming from, but it's quite continuous. "
;
```

These objects will respond to **smell smell of dampness** and **listen to sound of dripping water**, but will refuse just about any other command with a message like "You can't do that to a smell/sound." They effectively work as decoration objects, which means that if there was another object representing water in the cave, say, a command like **taste water** would be directed to the water without disambiguation.

Note that `Odor` and `Noise` objects won't be listed in response to intransitive smell and listen commands unless you explicitly give them `smellDesc` and `listenDesc` properties, in which case the `smellDesc` or `listenDesc` will be used in response to the intransitive smell or listen, while the `desc` will be used in response to the transitive **smell smell** or **listen to noise**. So, for example, in the above case we might better have written:

```
cave: Room 'Large Cave'
   "There's general smell of dampness here, as well as the sound of
     dripping water. "
;

+ Odor 'smell of dampness; pervasive musty; damp'
   "It's a pervasive, musty smell. "
   smellDesc = "There's a pervasive smell of damp in the cave. "
;

+ Noise 'sound of dripping water; continuous'
    "You can't locate where it's coming from, but it's quite continuous. "
    listenDesc = "You can hear a continuous dripping sound. "
;
```

It's not always appropriate to do this, however, especially when the smell has already been mentioned on another object. For example, if on an oven object we defined `smellDesc = "There's a smell of burning coming from the oven"`, we might well want to define an `Odor` object to represent the smell of burning, but we wouldn't then give that `Odor` object a `smellDesc` as well.

16.2   Sensory Connections

Sound, light and smells all travel. In the standard IF model, the player character can only sense what's in the same room as s/he is, but this may not always be realistic. We may, for example, have implemented a very large hall as two or more rooms, perhaps one room representing the east end and the other the west. Or we may have done the same for a town square or a large field or a long corridor. In such cases, whereas it's perfectly reasonable that the player character cannot taste or touch anything in the other rooms, it's not so reasonable that s/he cannot see, hear or smell them, since in real life you'd be able to see the other end of the hall, the other corners of the square, the other parts of the field and so on. Ideally, we need a mechanism to provide sensory connections between rooms in such situations.

In adv3Lite this mechanism is provided by the **SenseRegion** class. A **SenseRegion** is a special kind of **Region**; in other words its something that several rooms can be defined as belonging to, in just the same way as they can be defined as belonging to any other kind of Region. But unlike the basic **Region** class, a **SenseRegion** provides sensory interconnections (by default for sight, sound and smell) across all the rooms it contains. You set up a **SenseRegion** in just the same way as you'd set up an ordinary **Region**, except that you declare it to be of the **SenseRegion** class. For example to create a large hall comprising two rooms with a sensory connection between the two ends of the hall you could do this:

```
hallRegion: SenseRegion
;

eastHall: Room 'Hall (east)'  'east end of the hall'
   "This large hall continues to the west. "
   west = westHall
   regions = [hallRegion]
;

westHall: Room 'Hall (west)'  'west end of the hall'
   "This large hall continues to the east. "
   east = eastHall
   regions = [hallRegion]
;
```

We can customize the particular connections any given **SenseRegion** allows by overriding the following properties on it:

- **canSeeAcross** – (true by default); is it possible to see something in one room in this SenseRegion from another room in this SenseRegion?

- **canHearAcross** – (true by default); is it possible to hear something in one room in this SenseRegion from another room in this SenseRegion?

- **canSmellAcross** – (true by default); is it possible to smell something in one room in this SenseRegion from another room in this SenseRegion?

- **`canThrowAcross`** – (nil by default); is it possible to throw something from one room in this SenseRegion at a target in another room in this SenseRegion? (If not the projectile will fall short into the room it was thrown from).

- **`canTalkAcross`** – (nil by default); is it possible for an actor in one room in this SenseRegion to talk to an actor in another room in this SenseRegion?

These properties are global, all-or-nothing, in that they allow either everything or nothing to be seen, heard or smelled in remote locations. This may not always be realistic; one might see a large scarecrow at the other end of a field, but not a small buttercup; one might smell a bonfire over the fence, but not a rose; one might hear the sound of drumming from the far end of the hall but not the ticking of a quiet clock. Adv3Lite allows us to model these situations by defining the remote sensory properties of individual objects.

The first way we can do this is via the **`sightSize`**, **`soundSize`** and **`smellSize`** properties of a Thing, each of which can be **`small`**, **`medium`** or **`large`** (the default is **`medium`**). How these properties behave depends on what else we do. Supposing we don't override or define any of the methods we're about to discuss below, then the default behaviour will be:

- **`small`** – the object cannot be seen/heard/smelt from a remote location.

- **`medium`** – the object can be detected in the relevant sense, but no detail can be made out (it will be described as 'too far away to make out any detail' or 'too far away to hear/smell distinctly).

- **`large`** – the object can be detected in the relevant sense, and its normal desc, listenDesc or smellDesc will be shown in response to an attempt to examine, listen to, or smell it.

However, there's two more series of methods that may further affect this behaviour. Firstly, the set of methods (defined on Thing) that determine whether a particular object can be detected by a given sense:

- **`isVisibleFrom(pov)`** – Is this object visible to pov? By default this is true if sightSize is not small, but can be overridden to give different results.

- **`isAudibleFrom(pov)`** – Is this object audible to pov? By default this is true is soundSize is not small, but can be overridden to give different results.

- **`isSmellableFrom(pov)`** – Is this object smellable to pov? By default this is true if smellSize if not small, but can be overridden to give different results.

- **`isReadableFrom(pov)`** – Can pov read what's written on this object? By default this is true only if sightSize is large, but can be overridden to give different results.

In each case the *pov* parameter is the point-of-view object, i.e. the actor doing the

sensing (normally the player character). Each of these methods could therefore test for the location of the pov object before deciding whether the object they're defined on can be sensed from there. In other words, once we've defined a number of rooms as belonging to the same `SenseRegion` we can further refine which objects can be seen, heard, smelt or read from particular rooms within that `SenseRegion`. This probably works best on objects that are fixed in place, so that we can be sure which room it's in when we're writing isXxxableFrom(pov) routines to determine where it can be sensed from. Whereas it would be possible to do this with portable objects, it's probably easier to stick just to using their `sightSize`, `smellSize` and `soundSize` properties unless we really do need finer-grained control.

The other set of methods that can affect the behaviour of these properties are the remoteXxxDesc methods, namely:

- `remoteDesc(pov)` – the description of this object when examined from pov

- `remoteListenDesc(pov)` – the description of this object when listened to from pov

- `remoteSmellDesc(pov)` – the description of this object when smelt from pov

Once again *pov* is the point-of-view object doing the sensing, normally the player character. These three methods can thus be used to describe what an object looks like, sounds like or smells like from a remote point-of-view. If any of these methods is defined on an object, it will be used regardless of the object's `sightSize`, `soundSize` or `smellSize` (as the case may be), provided that the corresponding `isVisibleFrom(pov)`, `isAudibleFrom(pov)` or `isSmellableFrom(pov)` method returns true (which, by default, they will if the corresponding size is not `small`).

16.3 Describing Things in Remote Locations

The properties and methods we have just discussed determine what can be sensed from a remote location and how remote objects are described when they are examined, listened to or smelled. One further point that remains to be discussed is how objects are listed in remote locations.

You may recall that 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`. 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 `remoteSpecialDesc(pov)`, where *pov* is the actor doing the viewing (usually the player character). Provided an `initSpecialDesc` is defined we can also correspondingly define `remoteInitSpecialDesc(pov)`. In both cases the remote version will be used when

the actor and the object being listed are in different top-level rooms, although their default behaviour is simply to use `specialDesc` and `initSpecialDesc`.

In the case of an Actor, the `remoteSpecialDesc(pov)` method is farmed out to the current ActorState, if there is one, otherwise `actorRemoteSpecialDesc(pov)` is used; this parallels the use of `specialDesc` on an Actor.

Where neighbouring locations are connected by a SenseRegion, we need to use `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'
    specialDesc = "A large wooden table occupies much of the floor-space at
      this end of the room. "
    remoteSpecialDesc(pov)
    {
       switch(pov.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;
       }
    }
;
```

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 in the same SenseRegion, 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 a number of ways we can fix this. The first and simplest is to give rooms with award titles like these a name property as well as a roomTitle property, for example:

```
hallEast: Room 'Hall (east)' 'east end of the hall'
;
```

This would suffice to give us the much improved:

In the east end of the hall, you see a rubber duck.

For slightly greater control, we could override the `inRoomName(pov)` method of the remote room, for example:

```
hallEast: Room 'Hall (east)' 'east end of the hall'
   inRoomName(pov) { return 'at the far end of the hall'; }
;
```

Which would give us the arguably even better (at least where there are only two rooms in the SenseRegion):

At the far end of the hall, you see a rubber duck.

On the other hand, if we had a long road with north, mid, and south sections, then we might well use `inRoomName(pov)` on the middle section to vary the name according to where we were being viewed from:

```
roadMid: Room '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 <<if pov.isIn(roadNorth)>>south
       <<else>>north<<end>>';
    }
;
```

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 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 `remoteContentsLister` on the room we're looking at. 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 as the argument 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)
    remoteContentsLister =
        new CustomRoomLister('Right at the far end of the hall {i}
            {catch} sight of ');

;
```

Note that if we do this, whatever we define here will take precedence over whatever we did with `inRoomName(pov)` on the remote room.

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", we have to work a little harder. The constructor to `CustomRoomLister` can call a number of optional named arguments, one of which can be the definition of the method we want to use to prefix the list with. So we could do something like this:

```
hall: Room 'Hall'
    west = startRoom
    regions = [hallRegion]
    inRoomName(pov) { return 'at the far end of the hall'; }
    remoteContentsLister =
        new CustomRoomLister('', prefixMethod: method(lst, pl, irName)
            { "At the far end of the hall <<if pl>>are <<else>>
                is<<end>> "; });
;
```

In brief `prefixMethod:` means that we're passing the name `prefixMethod` parameter here. This needs to be passed as a floating method, which is effectively defined here with the following syntax:

```
method(lst, pl, irName)
{
    "At the far end of the hall <<if pl>>are <<else>> is<<end>> ";
}
```

The `prefixMethod` for `CustomRoomLister` must be a method that takes three arguments: *lst* – the list of objects to be listed; *pl* – whether the list is grammatically singular or plural (true means it's plural) and *irName*, the `inRoomName(pov)` of the room it's being called on. Here we simply use the pl parameter to determine whether the list of objects is grammatically singular or plural (it's plural either if there's more than one item in the list, or if the first item in the list is itself plural, e.g. 'flowers') and then to use 'are' or 'is' in the prefix text accordingly.

If this seems a bit baffling at first, don't worry about it for now, it *is* quite an advanced technique, but it is at least worth being aware of.

There are a further pair of optional named parameters that can be passed to the CustomRoomLister constructor: `suffix` and `suffixMethod`. The former of these is just a piece of text in a singe-quoted string that appears at the end of the list in place of a plain full stop (or period). The latter is a floating method to append text to the end of a list, for example:

```
hall: Room 'Hall'
    west = startRoom
    regions = [hallRegion]
```

```
    inRoomName(pov) { return 'at the far end of the hall'; }
    remoteContentsLister =
        new CustomRoomLister('\^', suffixMethod: method(lst, pl, irName)
        {
          " <<if pl>>are <<else>> is<<end>> <<irName>>. ";
        });
;
```

Which might yield:

A rubber duck is at the far end of the hall.

Note in this case how we passed a prefix string of '\^' to ensure that the resultant sentence would start with a capital letter.

16.4   Remote Communications

There's one further type of remote sensing that can sometimes turn up in Interactive Fiction, and that's where we want the player character to talk to another actor via telephone or videophone, particularly in a game set in the modern world where the protagonist is quite likely to be carrying a mobile phone (or cell phone). This situation would be awkward to model using a SenseRegion, so we instead use the `commLink` object for this kind of remote communication.

Using `commLink` is quite straightforward. Mostly, we just call one or other of its four commonly used methods:

- `connectTo(other)` – establish an audio-link between the player character and *other*, where *other* will normally be another Actor. If this method is called as `connectTo(other, true)`, then we establish an audio-visual link with the other actor.

- `disconnect()` - disconnect all audio and audio-visual links between the player character and other actors.

- `disconnectFrom(other)` – disconnect the audio or audio-visual link with *other* leaving any other communications links in place. The *other* parameter may be supplied as a single actor or as a list of actors.

- `isConnectedTo(other)` – determines whether the player character currently has a communications link to *other*; returns nil if not, or `AudioLink` if there's an audio link, or `VideoLink` if there's an audio-visual link.

For example, a fairly basic implementation of a PHONE command that allows the player character to call other known actors using his/her mobile phone might look like this:

```
DefineTAction(Phone)
    addExtraScopeItems(role)
    {
        inherited(role);

        scopeList = scopeList.appendUnique(Q.knownScopeList.subset({x:
            x.ofKind(Actor)}));
    }
;

VerbRule(Phone)
    ('phone' | 'call') singleDobj
    : VerbProduction
    action = Phone
    verbPhrase = 'phone/phoning (whom)'
    missinqQ = 'who do you want to phone'
;

modify Thing
    dobjFor(Phone)
    {
        verify()
        {
            illogical('{I} {can\'t} phone {that dobj}. ');
        }
    }
;

modify Actor
    dobjFor(Phone)
    {
        preCond = new ObjectPreCondition(mobilePhone, objHeld)

        verify()
        {
            if(commLink.isConnectedTo(self))
                illogicalNow('{I} {am} already talking to {the dobj} on the
                    phone. ');
        }

        action()
        {
            commLink.connectTo(self);
            sayHello();
        }
    }

    endConversation(reason)
    {
        /* Don't end the conversation if we're on the phone and we walk about */
        if(commLink.isConnectedTo(self) && reason == endConvLeave)
            return;

        if(canEndConversation(reason))
        {
            sayGoodbye(reason);
            commLink.disconnectFrom(self);
        }

        /*
         *    otherwise if the player char is about to depart and the actor won't
```

```
         *   let the conversation end, block the travel
         */
     else if(reason == endConvLeave)
         exit;
   }
;
```

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   What Attachable Means

In adv3Lite 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 IF, 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 adv3Lite library can scarcely define a straightforward Attachable class that works absolutely fine straight out of the box; instead it provides a number of classes to deal with the simpler cases. In adv3Lite the attachable relationship is always asymmetric and usually many-to-one. This means that in a command like **attach x to y**, x is regarded as something that is moved to the vicinity of y in order to be attached to it, and many different things can be attached to y, without the reverse being true. For example, I could attach many small metal objects to a magnet, and I could then attach the magnet to the fridge, but I could not attach the magnet to many objects at once (though there may be many objects attached to the magnet).

Hopefully this will all become clearer when we look at each of the Attachable classes in turn.

17.2   SimpleAttachable

SimpleAttachable is the principal Attachable class defined in the adv3Lite library, in that all the other types of Attachable descend from it or depend on it in some way. The **SimpleAttachable** class is meant to make handling one common case easy, 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 **allowableAttachments** property on the major **SimpleAttachable** to hold a list of

items that can be attached to it, e.g.:

```
allowableAttachments = [redMagnet, blueMagnet]
```

If a more complex way of deciding what can be attached to a major `SimpleAttachable` is required, override its `allowAttach(obj)` method instead, so that it returns true for any *obj* that can be attached, e.g.:

```
allowAttach(obj) { return obj.ofKind(Magnet); }
```

The other properties and methods of SimpleAttachable you may want to use from time to time include:

- `attachments` – a list of the objects that are currently attached to me

- `attachedTo` – the object I'm currently attached to

- `isAttachedToMe(obj)` – returns true if *obj* is attached to me and nil otherwise

- `isAttachedTo(obj)` – returns true either if *obj* is attached to me or I'm attached to *obj*.

- `isFirmAttachment` – if true (the default) then I must be detached from anything I'm attached to before I can be moved

- `allowOtherToMoveWhileAttached` – if and only if true (the default) then the object I'm attached to can be moved while I'm still attached to it.

Remember the distinction to the objects I'm attached to and the objects attached to me. If I'm a magnet then I may be attached to a fridge while having any number of paper-clips attached to me.

One other thing the SimpleAttachable class does is to treat **fasten** and **unfasten** as equivalent to **attach** and **detach**, since in this case it's hard to see what the distinction between attaching and fastening could be.

17.3 NearbyAttachable

A common requirement with Attachables is that if we attach one object to another, the two objects should be in the same place (that is, within the same container). The `NearbyAttachable` class is a special kind of `SimpleAttachable` that enforces this condition. More specifically it enforces:

- When one `NearbyAttachable` is attached to another, the item that's being attached is moved to the location of the object it's being attached to.

- When one `NearbyAttachable` is detached from another it ends up still in the same location (the location of the object it's just been detached from).

- If the object a `NearbyAttachable` is attached to is moved, that object is moved to the location of the object to which it's attached.

If **NearbyAttachable** 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 **attachedLocation** and **detachedLocation** properties that define where the **NearbyAttachable** should end up when its attached to or detached from another object. For example, suppose we want a LegoBrick class so that if the player character is detached one brick from another, the detached brick ends up being held by the player character:

```
class LegoBrick: NearbyAttachable
    canAttachTo(obj) { return obj.ofKind(LegoBrick); }
    detachedLocation = gActor
;
```

A more common use of **NearbyAttachable** might be to join two lengths of cable together, for example. When a first length of cable is attached to a second, it ends up in the same location as the second. If we detach the first length of cable again, it's no longer attached (obviously), but it doesn't move to a new location. For this kind of situation NearbyAttachable should work straight out of the box without any further customization.

17.4   AttachableComponent

An **Attachablecomponent** is a **SimpleAttachable** that represents a part of the object to which it's attached, for example the handle of a knife. We could often use a simple **Component** to do this job, but the potential advantage of using an **AttachableComponent** is that if it is detachable by some means (unscrewing, perhaps) it automatically becomes portable, and that if we like we can make it readily re-attachable. For example, suppose we want a mobile phone which has a button which can be unscrewed and reattached; an outline implementation might look like this:

```
+ mobilePhone: SimpleAttachable 'mobile phone; cell; cellphone'
    allowableAttachments = [button]
;

++ button: AttachableComponent, Button 'button'
    isUnscrewable = (attachedTo == mobilePhone)

    dobjFor(Unscrew)
    {
        action()
        {
            "{I} unscrew{s/ed} the button from the mobile phone. ";
            detachFrom(location);
            actionMoveInto(gActor);
        }
    }
;
```

Note that this is a rare instance where the order of classes in the declaration of an object is important. In this case **AttachableComponent** *must* come before **Button** in the declaration of the button object to ensure that we get AttachableComponent's version of the **isFixed** property (which switches between nil and true according to whether or not the button is detached). Note also that we need to specify button in the **allowableAttachments** list of the **mobilePhone** object to ensure that we can reattach it.

17.5   Attachable

The **SimpleAttachable** class in adv3Lite is designed to be easy to use in a number of common cases, but there are a few cases it's not so well equipped to handle, in particular where we want to be able to attach the same object to more than one other thing (as opposed to having several things attached to it). Examples might include a length of cable used to make an electrical connection between two separate objects, or a rope used to tie two mountaineers together. Although it may sometimes be possible to shoehorn such cases into the many-to-one relationship allowed by a **SimpleAttachable**, sometimes it may not, and other times it may be awkward to do so since it could lead to a counter-intuitive order of attachment (**attach outlet to cable** instead of the more intuitive **attach cable to outlet** for instance). To allow for such cases adv3Lite provides an **Attachable** class, which can be used in many-to-many attachment relationships. **Attachable** inherits directly from **NearbyAttachable**, and works in a similar way, but instead of making use of an **attachedTo** property that points to the single thing it's attached to, it uses an **attachedToList** property that contains list of the things it's attached to (which remains distinct from the **attachments** property that contain a list of the items attached to it – remember that attachment is always considered asymmetric in adv3Lite). The Attachable class also defines the following additional properties:

- **maxAttachedTo**: the maximum number of items to which this object can be attached at once. The default value is 2, on the basis that this is likely to be the most common case. If you don't want there to be any limit at all, you can set this property to nil. Note that the **maxAttachedTo** property does not impose any limit on the number of items that can be attached to the objects it's defined on (the number of items in the **attachments** property); it merely limits the maximum length of the **attachedToList** property.

- **reverseConnect(obj)**: If this method returns true for *obj* then this reverses the order of connection between self and *obj* when *obj* is the direct object of the command; i.e. **connect obj to self**, will be treated as **connect self to obj**. Since the adv3Lite attachment relationship is asymmetric, this method can be used to help enforce the attachment hierarchy intended by the game author.

- **multiPluggable**: if this property is true (the default) then this allows an **Attachable** that's also a **PlugAttachable** to be plugged into more than one thing at a time (e.g. to implement a cable that's needed to make an electrical connection between two different items).

17.6 PlugAttachable

PlugAttachable is a mix-in class we can use with **SimpleAttachable, NearbyAttachable** or **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: PlugAttachable, NearbyAttachable, Fixture 'socket'
    allowableAttachments = [plug]
;

+ plug: PlugAttachable, NearbyAttachable 'plug'

;
```

Often this is all that is needed. Note, however, that since **PlugAttachable** is a mix-in class it must come first in the class list.

There are a few things we can tweak if we need to. The **socketCapacity** property on the socket object (more generally, the object plugged into) controls how many objects can be plugged into it at once; the default is 1. The **needsExplicitSocket** property (by default true) specifies whether a **plug in** command needs to specify a socket. If this is nil, the player can issue a command like **plug in the tv** without needing to specify what to plug the tv into (and without the game even needing to define a corresponding socket object).

Exercise 22: The player character is the only survivor aboard a small scouting spaceship 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 repaired, 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.

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 adv3Lite 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 your 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 adv3Lite 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 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 relate to 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).

- `openWhenExamined` – the goal becomes open when this object has been described (i.e. when the player has examined it).

- `openWhenMoved` – the goal becomes open when this object has been moved

- `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'
    openWhenExamined = 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`, `closeWhenExamined`, `closeWhenMoved`, `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 had been opened, then we could modify Goal to allow this:

```
modify Goal
    openWhenOpened = nil
    closeWhenOpened = nil
    openWhen = (inherited || (openWhenOpened != nil && openWhenOpened.opened))
    closeWhen = (inherited || (closeWhenOpened != nil && closeWhenOpened.opened))
;
```

Then we'd be able to use our new `openWhenOpened` and `closeWhenOpened` 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";
```

Calling **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 **libScore.showScoreRankMessage(msg)**, e.g.:

```
modify libScore
    showScoreRankMessage(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.

19   Beyond the Basics

19.1   Introduction

We've now covered all the basics of writing Interactive Fiction in TADS 3 with adv3Lite; 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/adv3Lite programming tasks 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 and adv3Lite 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);
```

There's seldom any need to do this in adv3Lite, but it's worth knowing that the Parser deals with results of tokenizing the player input. For some information on what the parser does 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 adv3Lite without understanding that part of the system in any depth.

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*.

19.2.2   Object Resolution

The adv3Lite 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", and its parser isn't fussy about word order, making it easy to match noun phrases like 'Cranky the Clown' or 'S and P Magazine'. We've also seen that we can bracket selected words in the **vocab** 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).

If the order of words is important, or you want an item to match a particular phrase but not necessarily the individual words that make up that phrase, you can use the **matchPhrases** property to define one or more phrases you want the object to match; this can be defined as a single-quoted string or as list of single-quoted strings.

For example, suppose we have a game that contains both a red wine bottle and a red bottle. The phrase 'red bottle' is then ambiguous, and it's not easy to see how the player could refer to the red bottle as distinct from the red wine bottle. One way to handle this would be to define the **matchPhrases** property on the red wine bottle:

```
redWineBottle: Thing 'red wine bottle; glass'
   matchPhrases = ['red wine', 'wine']
;

redBottle: Thing 'red bottle; glass'
;
```

The `redWineBottle` object will now match 'red wine bottle' or 'wine bottle' but not 'red bottle'. Note that the `matchPhrases` property doesn't add any vocab to an object; it just restricts the circumstances under which its existing vocab words will match. Note also that we need to list 'wine' in the `matchPhrases` list, since the presence of 'red wine' in the list would otherwise not allow 'wine' to match by itself.

If this still 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 `Mentionable`) 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 `Thing`) in the *Library Reference Manual*. For even finer-grained control you could override `scoreObject()` on a particular Thing or class; again, for further details, consult 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. There are various ways of adjusting scope in adv3Lite. The most fundamental is probably the `addExtraScopeItems(role)` method of a particular action; by default this simply calls

addExtraScopeItems(action) on the actor's current room (which thus provides another opportunity to affect scope), which in turn calls **addExtraScopeItems()** on all the regions that room is in. By default the methods on Room and Region simply add to scope any objects that are in the Room or Region's **extraScopeItems** list.

For example, the GoTo Action, which allows the player character to travel to the location of any known room or object with a **go to wherever** command defines its **addExtraScopeItems** method like this:

```
/* Add all known items to scope */
    addExtraScopeItems(whichRole?)
    {
        scopeList = scopeList.appendUnique(Q.knownScopeList);
    }
```

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 **vocab** property of the object.

If we want we can define the order in which items are listed in a disambiguation prompt using the **disambigOrder** property, which by default takes its value from the **listOrder** 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. Adv3Lite treats items as effectively indistinguishable if they have the same names and can be referred to by the same vocab. So, for example, to define a bowl containing five identical grapes we might write:

```
class Grape: Food 'grape; green round; fruit[pl]'  'grape'
    "It's round and green. "
;

bowl: Container '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. The adv3Lite library knows about most of the common ones already, but if for some reason your game objects include several pericopae (singular pericope), you would stipulate this uncommon plural in braces immediately after the singular form:

```
pericope: Thing 'short pericope{pericopae}; brief'
;
```

Similarly, if your game featured alien insects called mantikae (singular mantika) you could just include the plural suffix in curly braces:

```
mantika: Thing 'black mantika{-e}; alien; insect'
;
```

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.

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`, `nominalOwner()` and `ownsContents` properties on Thing in the *Library Reference Manual*.

We have seen how identically-named objects are automatically grouped together by adv3Lite. It can also happen that similarly-named objects are arranged in a class hierarchy, so that, for example, we have a Coin class from which the GoldCoin and

SilverCoin classes inherit. In such cases we can, if we like, inherit part of the vocab and/or name. For example, a + sign in the name part of a vocab string indicates that the name part of the superclass is to be inserted at that point; for example:

```
class Coin: Thing 'coin; round metal'
;

class GoldCoin: Coin 'gold +'
;

class SilverCoin: Coin 'silver +'
;

smallSilverCoin:SilverCoin 'small +'
;
```

In this case, the GoldCoin and SilverCoin classes have name properties of 'gold coin' and 'silver coin' respectively, while the smallSilverCoin's name will be initialized to 'small silver coin'. In addition, all these classes and objects will inherited the adjectives 'round' and 'metal' from the Coin class. If we don't want adjectives to be inherited from a parent class, we can start the adjectives section of the vocab property with a minus sign, like so:

```
squarePlasticCoin: Coin ' square plastic +; - blue'
;
```

19.4   Fancier Output

We have already seen how we can use HTML tags to format the output of a game; for example "..." 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*.

If you want to use any of these fancy output features (HTML mark-up, sound, and graphics) 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*.

Amongst other things 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.

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 if we like we can change both the person and the tense. In addition to second-person narration, adv3iLte 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); or, indeed, in any of four other tenses (perfect, past perfect, future and future perfect), though these will probably be less commonly used.

To change to first-person or third-person narration is fairly straightforward; we just need to override the `person` property on the player character object (typically called `me`). The default value is `2`; for first-person narration we simply change it to `1`, and for third-person narration we change it to `3`.

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; her'
    person = 3
;
```

If we have written all our own action response messages in the form "{I} turn{s/ed} the handle" rather than "You turn the handle", then they will automatically adapt to whichever person and tense 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 **person**; 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, then it will have been as well to use parameter substitutions for all our custom messages so that they'll change tense automatically too. Then we can change tense by changing the value of **gameMain.usePastTense** between nil and true, or, if we want to use one of the more exotic tenses, by changing the value of **Narrator.tense** (but if we do that once, we must keep on doing it rather than relying on **gameMain.usePastTense**). If we're only changing between past and present (the most likely combination) we can also use the tSel() macro or a vertical bar in a message parameter substitution to write text that changes text accordingly. The following two strings, illustrating the use of these two methods, are effectively equivalent:

```
"There <<tSel('is', 'was')>> absolutely nothing <<tSel('here', 'there')>>. ";
"There {is|was} absolutely nothing {here|there}. ";
```

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.getOutermostRoom.lookAroundWithin();
```

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 Room:

```
modify Room
    statusName(actor)
    {
        inherited(actor);
        if(libGlobal.playerCharName != nil)
```

```
            " (as <<libGlobal.playerCharName>>) ";
    }
;
```

We have to use **libGlobal.playerCharName** 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). If an Actor can switch between being the player character and an NPC, when we need to use its **isPlayerChar** property to adjust its description between perspectives accordingly, for example:

```
bob: Actor 'Bob; tall gangling; man fellow; him'
   "<<if isPlayerChar>>You're a tall man, and not bad-looking, though you
     say it yourself. <<else>>He's a gangling sort of fellow who rather
   fancies himself. <<end>>"

    person = 2
;
```

Note that even if Bob starts out by being the player character, if he may later become an NPC you need to make him an Actor and define his **vocab** property in a way that will be suitable for his future role as an NPC.

19.6   Pathfinding

Adv3Lite comes with a built-in pathfinding module. We have already seen how to use it to calculate a route for an NPC to take from his/her current location to any given destination (provided a viable route exists), using **routeFinder.findPath(start, destination)**. For a full discussion, see section 14.12 above. The default behaviour of the **routeFinder** object is to exclude any route that passes through a locked door (on the basis that an NPC would probably be unable to navigate it). To change that assumption you can set **routeFinder.excludeLockedDoors** to nil.

There's also a **pcRouteFinder** object that can be used to help the player character navigate the map using **go to** and **continue** commands. The difference between this and the **routeFinder** is that it will only find routes to places (or objects) the player character already knows about via TravelConnectors the destinations of which are already known to the player character. This prevents players from using the route-finder to avoid the need to explore the map. However, if there's an area of the map that should be familiar to the player character at the start of the game (such as her own house or his home town), then you can define that area as a **Region** and then define **familiar = true** on that Region. The player will then be able to navigate the familiar region with **go to** and **continue** without first having to explore it.

Beyond that, there's nothing more you need to do to make this form of player navigation work; it's built in as standard. For example, the player can enter the command **go to kitchen** to take the first step towards the kitchen, and then on subsequent turns enter the command **continue** or just **c** to keep taking the next step until the player character arrives at the destination. This mechanism allows the player to stop along the route to interact with anything that turns up, and then continue the journey if and when appropriate.

If you want to disable this kind of navigation for the player, the simplest thing to do is simply to exclude pathfind.t from your build. If, however, you need to include pathfind.t either because you want to use the `routeFinder` for NPCs or because you only want to disable the **go to** command under particular circumstances, you could use a Doer:

```
Doer 'go to Thing'
    execAction(c)
    {
        "In this game, you'll have to navigate with compass directions. ";
        abort;
    }
;
```

Both `routeFinder` and `pcRouteFinder` are designed to find routes around a game map. They both, however, inherit from the more abstract `Pathfinder` class which could in principle be used as the basis for other types of path-finding. To to this you would need to define your own object of the `Pathfinder` class and define its `findDestinations(cur)` method to find all the destinations one step away from *cur*, where *cur* represents a particular step along the path, which would be given as a two element list, the second element of which is the node to which it leads and the first element defines how to get to this node from the previous one. To explain how to go about this in any more depth is way beyond the scope of this book, but the place to start would be to examine how `routeFinder` and `pcRouteFinder` implement this method.

19.7   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, if only to point out which other parts of the *System Manual* are most worth studying.

19.7.1   Varying, Optional and Named 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, including 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*.

A related but not identical case is where we want a function or method to have optional parameters. A varying argument list is one in which there can be any number of parameters. With optional parameters, on the other hand, the number of parameters is fixed, but some of them can be omitted when the method or function is called. This can be particularly useful for parameters that usually take a default value or which are commonly not used.

We define an optional parameter by following it with a question-mark (?) in the argument list, for example:

```
truncate(str, len, upperCase?)
{
    str = str.substr(1, len);
    if(upperCase)
        str = str.toUpper();
    return str;
}
```

This returns a string that is **str** truncated to the first *len* characters and optionally converted to upper case. If we don't want to convert the string to upper case, we needn't supply the third parameter at all; the function could simply be called as:

```
  msg = truncate('The rain in Spain stays mainly in the plain', 16);
```

If an optional parameter is not used in a function or method call, its value at the start of the function or method is nil. If we want it to default to some other value, we can initialize it by following it with = plus the required default value, e.g.:

```
increment(x, y = 1)
    return x + y;
;
```

If this is called as **increment(2)** it will return 3; if it is called as **increment(2, 2)**, it will return 4.

The above examples have mixed compulsory and optional parameters. It's also perfectly legal to have a function or method with only optional parameters (and no compulsory ones), but where there is a mix of compulsory and optional parameters, the compulsory ones must come first. For the full story on optional parameters see the chapter on 'Optional Parameters' in Part III of the *TADS 3 System Manual*.

A final variation on the kinds of argument list we can create is *named arguments*. This allows an argument to be passed by name instead of positionally. We indicate a names argument by following its name with a colon. For example we could write:

```
truncate(str:, len:, upperCase:?)
{
    str = str.substr(1, len);
    if(upperCase)
        str = str.toUpper();
    return str;
}
```

Since the arguments are named, they don't need be listed in the same order when the function or method is called. For example, we could call the above function with:

```
local msg = truncate(len:6, upperCase: true, str: 'oranges and lemons');
```

For the full story on named arguments see the chapter on 'Named Arguments' in Part III of the *TADS 3 System Manual*.

19.7.2   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, however. 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.

19.7.3   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.*

19.7.4   Multi-Methods

TADS 3 also has a 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);
    "{I} {put} <<obj.theName>> into <<cont.theName>>";
}

putIn(Liquid liq, Vessel ves)
{
    local blk = liq.bulk;
    liq.bulk -= ves.getFreeBulk();
    if(liq.bulk <= 0)
        liq.moveInto(nil);
    vess.addLiquid(liq, blk);
    "{I} pour{s/ed} <<liq.theName>> into <<ves.theName>>. ";
}
```

For more details, see the chapter on Multi-Methods in Part III of the *System Manual*.

19.7.5   Modifying Code at Run-Time

Not only can we change the data held in a property at run-time (we'd hardly be able to do much in TADS 3 if we couldn't), we can also change the code attached to methods. That is, we can assign a new method to an object property (and also retrieve the method that's attached to a property for use elsewhere). We do this using the methods `getMethod(prop)` and `setMethod(prop, meth)`, which we encountered earlier in relation to double-quoted strings. For the full story on these two methods see the chapter on TadsObject in Part IV of the *TADS 3 System Manual*.

In order to be able to do more than copy a method from one object property and apply it to another, we need some means of defining a method which can later be attached to a property. This can be done either as a *floating method* or as an *anonymous method*.

A *floating method* is so called because it doesn't belong to any particular object. We can create it using the keyword `method` in much the same way that we create a function. For example:

```
method describeMe
{
    if(ofKind(NonPortable))
        "\^<<theName>> is not the sort of thing you could carry
            around with you. ";
    else
        "It looks small enough to be carried. ";
}
```

This method could then be attached to, say, the desc property of an object using a statement like:

```
    ballBearing.setMethod(&desc, describeMe);
```

An anonymous method is created in a way similar to that in which we'd create an anonymous function, for example:

```
    local meth = method(obj, newCont) {
        gMessageParams(obj, newCont);
        if(obj.bulk > maxBulk)
        {
            "{The subj obj} {is} too big to fit into {the newCont}. ";
            exit;
        }
        else
            "{The subj onj} {is} being inserted into {the newCont}. ";
    }

    blackBox.setMethod(&notifyInsert, meth);
```

Note that in both cases the floating or anonymous method takes on the context of the object to which it's attached by `setMethod`. That is, the floating or anonymous method can refer to `self` and to methods and properties of the self object (and can use keywords such as `inherited` and `delegated`) and this will all work as expected in the context of the object to which the anonymous/floating method has been attached.

We've already seen an example of this, where the library defines a CustomRoomLister class allows prefix and suffix *methods* to be passed via its constructor:

```
class CustomRoomLister: ItemLister

    /* is the object listed in a LOOK AROUND description? */
    listed(obj) { return obj.lookListed && !obj.isHidden; }

    /*
     *   In the simple form of the constructor, we just supply a string that
     *   will form the prefix string for the lister. In the more sophisticated
     *   form we can supply an additional argument that's an anonymous method or
     *   function that's used to show the list prefix or suffix, or else just
     *   the suffix string.
     */
    construct(prefix, prefixMethod:?, suffix:?, suffixMethod:?)
    {
        prefix_ = prefix;

        if(prefixMethod != nil)
            setMethod(&showListPrefix, prefixMethod);

        if(suffix != nil)
            suffix_ = suffix;

        if(suffixMethod != nil)
            setMethod(&showListSuffix, suffixMethod);
    }

    prefix_ = nil
    suffix_ = '. '

    showListPrefix(lst, pl, irName)
```

```
    {
        "<.p><<prefix_>> ";
    }

    showListSuffix(lst, pl, irName)
    {
        "<<suffix_>>";
    }

    showSubListing = (gameMain.useParentheticalListing)
;
```

We could then pass the methods we want to use when we create the lister, for example:

```
longPath: Room 'Path'
    "This long path goes nowhere in particular. There's a market just to the
    west. <<first time>>As paths go it's fairly futile.<<only>> There's a
    field to the east. "
    east = field
    west = startroom
    remoteRoomContentsLister(other)
    {
        return new CustRoomLister(nil, prefixMethod:
            method(lst, pl, irName)
            { "Lying around <<irName>> <<if pl>>are<<else>>is<<end>> "; }
        );
    }
;
```

The above example incidentally shows that an anonymous method can be declared on the fly within a method or function call, just like an anonymous function. For further details see the section on 'Floating Methods' in the chapter on 'Procedural Code', and the section on 'Anonymous Methods' in the chapter on 'Anonymous Functions', both in Part III of the *Tads 3 System Manual*.

If anonymous and floating methods don't give you enough flexibility, you can go a step or two further with *DynamicFunc*. The DynamicFunc class lets you compile a string expression into executable code at run-time (specifically, into a function which you can then call). This string could, of course, be one that has been dynamically created elsewhere in your code, allowing a TADS 3 game to write some of its own source code and then compile it. For details, see the chapter on 'DynamicFunc' in Part IV of the *TADS 3 System Manual.*

19.8   Compiling for Web-Based Play

Full instructions for compiling and deploying a TADS 3 for playing on the web are given in Section VII of the *TADS 3 System Manual* ('Playing on the Web'), and the instructions specific to adv3Lite can be found in the *adv3Lite Library Manual*, so they need not be repeated here. A few points may be noted in passing, however.

First, a web-based game cannot use the Banner API (although it can display a

standard interpreter layout with a status line). So, if your game needs the Banner API, you'll have to compile and deploy it in the normal way (although to date, few if any published TADS 3 games have made much use of the Banner API). On the other hand a web-based game has full access to the features of the browser on which it's played, including Javascript, CSS and HTML DOM, which are not available through the standard HTML-TADS interpreter. This potentially allows a game author a great deal *more* control over the interface presented to players.

Second, if you wish to use the standard interpreter layout with just a status line and scrolling play area, it's pretty straightforward to write a TADS 3 game which can then be compiled in two versions, for playing via a traditional interpreter *and* for playing via the web. To do this, it's probably easiest to develop and test the traditional interpreter-based version first, and then compile and deploy the web version once you're done (which just requires a few manual tweaks to the .t3m file).

Third, although there are very few compatibility issues when switching between the web and traditional versions of a TADS 3 game that uses the standard interface layout, there are one or two. The first is that the <ABOUTBOX> tag can't be used in the web-based version, which almost certainly renders the entire setAboutBox() method redundant in the web version. Perhaps the easiest way to deal with this in a game intended for both traditional and web-based play is to surround your setAboutBox statement with #ifdef ... #endif conditional compilation statements thus:

```
gameMain: GameMainDef
#ifndef TADS_INCLUDE_NET
    setAboutBox()
    {
        "<ABOUTBOX> <b><FONT FACE=Verdana,TADS-Sans COLOR=BLUE SIZE=+2>
        <<versionInfo.name>><FONT></b>\b
        <FONT FACE=Verdana, TADS-Sans><<versionInfo.byline>>\bVersion
        <<versionInfo.version>></FONT>\b
        <b><FONT SIZE=-1 FACE=Verdana,Arial,Sans-Serif><<versionInfo.htmlDesc>>
        </FONT></b></ABOUTBOX>";
    }
#endif

...
;
```

Actually, this is not strictly necessary in this case, since `gameMain.setAboutBox()` will never be called from a game compiled for the web interface, and so leaving it in would almost certainly be harmless. There may, however, be other parts of our code that we want to work differently in the web based and traditional interpreter-based versions, and this illustrates what is probably the neatest way of handling it.

Another compatibility issue you may run into is if the traditional version of your game uses an identifier that's used for a different purpose in the web-ui library. For example, an early version of the web-ui library used 'path' as the name of a property. Using 'path' as the name of an object (a room representing a path, perhaps) in game

code then resulted in series of error messages the game was compiled for the web version, even though the game worked fine in the traditional version. The solution was to change the name of the offending identifier (in the case of 'path' we might change it to 'longPath' for example) and then do a full recompile for debugging. A full recompile for debugging may in any case be necessary when switching between the traditional and web-based versions of the same game. (Note, by the time you read this the web-ui libraries may have been changed to avoid the clash with the identifier name 'path', so you may not encounter this particular incompatibility).

20   Where To Go From Here

If you've followed *Learning TADS 3 with adv3Lite* this far, you're well on your way to knowing all you need to know to write your own games in TADS 3 using the adv3Lite library. "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 outs of every nook and cranny of the TADS 3 language and adv3Lite library; there's probably no one in the known universe who has *that* kind of knowledge in their head. Knowing what you need to know is rather knowing enough to carry out the tasks you commonly carry out in TADS 3/adv3Lite 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 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/adv3Lite.

The next step (which you needn't take straight away) is probably to read through the *adv3Lite Manual* and the *TADS 3 System Manual*, or at least to skim them so you get a rough idea of what they contain so you can find it when you need it.

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 *Adv3Lite 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*, *adv3Lite 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!

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.

Eric Eve

Harris Manchester College, Oxford

November 2013

21 Alphabetical Index