GETTING STARTED IN TADS 3

(version 3.1)

A Beginner's Guide

By

ERIC EVE

CONTENTS

Preface

The first edition of *Getting Started in TADS 3* was an attempt by one (then fairly novice) TADS 3 user to help others with some experience of programming IF systems to get up to speed on TADS 3. Since then, it has grown and developed as a result both of feedback from people who have used it, and of its author's attempts to improve it. In its present form it is intended to be used as one possible starting point for anyone wishing to learn to write Interactive Fiction using TADS 3. Another possible starting point would be *Learning TADS 3*, which takes a rather different approach.

This guide has never been intended as a complete manual covering every aspect of the language and library; it is instead intended to introduce new users to the basics of the system. If you are new to TADS 3, the best way for you to get to grips with it is probably for you to work through this guide and gain a reasonably secure understanding of its contents before moving on to the other documentation that covers the language and library in more depth.

Once you are reasonably comfortable with what *Getting Started* has to teach, however, you will need to move on to other documentation (and practice using the system) to start gaining mastery. The *TADS 3 Tour Guide* provides a much more thorough overview of the TADS 3 library, and can be used as a second tutorial to follow this one, or you may prefer to follow the more systematic introduction provided by *Learning TADS 3*. The *System Manual* describes the TADS 3 language in far more depth than is possible here, and you will find yourself needing to consult it frequently (reading through it at some stage might be no bad idea too, though there'll be parts you'll probably want to skip till you need them), while the *TADS 3 Library Reference Manual* provides a complete reference to the library. Finally, the *Technical Manual* covers a number of issues not dealt with in this guide, and goes into far more detail on some that are.

It remains only to say three things: first to record my appreciation for the enormous amount of effort that Mike Roberts has put into the development of TADS 3, and for his comments and suggestions on earlier drafts of this guide; second to record my thanks for his permission to incorporate parts of his own work into this guide; and finally to thank all those readers of previous versions of this guide for their feedback and suggestions, which will hopefully enhance its usefulness for those that follow.

Eric Eve
Harris Manchester College, Oxford
May 2012

Chapter One - Introduction

1. *General Introduction*

If you are reading this Beginner's Guide to using TADS 3 you will presumably already know what Interactive Fiction is, so I shall not offer an explanation here. You will presumably already have found your way to Mike Roberts's TADS pages (http://www.tads.org), will probably have downloaded the TADS 3 author's kit and tried it out, and may perhaps have some familiarity with another IF language such as TADS 2 or Inform. If you are still trying to work out how to install the TADS 3 compiler or do anything with it, the instructions in the section on "Creating your First TADS 3 Project" below should get you started. On the other hand, if you have already got going in TADS 3 you may want to skip the remaining sections of this chapter and maybe the next chapter as well. I shall not in any case spend a great deal of time explaining programming concepts such as functions, operators, statements, classes and objects, but the final two sections of this chapter provide a basic introduction to the way these are implemented in TADS 3 and to the programming concepts needed to follow this guide. Chapter Two then shows you how to implement a very short sample game; for some readers this may be the best place to start.

The approach taken in the remainder of this guide will be to introduce various features of TADS 3 through developing a more substantial game. Inform 6 users will recognize the genesis of this game in the *Inform Beginner's Guide* by Roger Firth and Sonja Kesserich (who have very kindly allowed me to borrow both Heidi and her forest), although rather than following the *IBG's* route of illustrating features of the language through three successive games, I shall stick with the same game, *The Further Adventures of Heidi*, and increase its complexity from chapter to chapter. The complete source code of the finished game, heidi.t, should be packaged in the zip file with this guide (if you downloaded it separately) or with the TADS 3 documentation set.

Chapter Three will explain in some detail how to define a room and a single object within that room for the game we'll be developing together throughout the remainder of the guide. Things will become rather more complicated in Chapter Four when we look at the use of various kinds of connector to move between rooms, and at the same time introduce a number of other features of the TADS 3 library and language so we can program a couple of basic puzzles (how to get to the top of a tree and find a diamond ring in a bird's nest). The main focus of Chapter Five will be the programming of an NPC (non-player character), in this case a friendly charcoal-burner who turns out to be the owner of the missing ring. Subsequent chapters will place a further series of obstacles between Heidi and the ring so that we can sample the excitements of lockable doors, dark caves, and a pair of conveniently buried Wellington boots (to mention a few) and thereby introduce a number of other features. By Chapter Seven we shall be sending Heidi on a boat trip down to the shops (well, a shop), and trying to figure out how to handle buying and selling. Chapter Eight will wrap things up with some suggestions on how to put some finishing touches to the game, a look at a different ways we could have tackled a couple of the problems we encountered, some suggestions on how to make testing and debugging slightly less painful, and some pointers on where to go next. Needless to say, the sample game won't be able to cover all the features of TADS 3, neither will it lay out the ones it does touch in a particularly systematic fashion; but it will introduce you to the process of writing Interactive Fiction in TADS 3 and in process of doing so it will introduce you to many of the features of the system you are likely to use in your own games.

It follows that this *Beginner's Guide* is not really intended for bedtime reading (although it may turn out to be an excellent cure for insomnia!); it is intended rather to be used sitting at your computer while you type the example code and see how it works when you compile it, and, even better, try experimenting and implementing things for yourself when invited to do so. If you'd like a sneak preview of where it's all going, by all means compile the heidi.t file that should have come with this *Guide* and play through the game – just don't expect it to be a masterpiece of IF!

This is by no means a formal manual, but I shall employ a few typographical conventions. Text in the Times New Roman font (like this) will be used for the bulk of the text. Text in **bold** will generally represent what a player might type at the TADS 3 command prompt when playing a game (e.g. **north** or **eat the baked banana**). Text in the Courier Font (`like this`) will be used for code or fragments of code (e.g. `name = 'Fred'`).

Note that both the *Guide* and the heidi.t file have been written to be compatible with TADS 3.1.0 (although most things will probably work well enough with earlier versions). If you have an earlier version of TADS 3 installed, please update it to the latest version before following the instructions in this guide, otherwise you may keep coming across things that do not work as they are described (as well as finding you get a raft of errors when you try to compile heidi.t). If by the time you read this you have a version of TADS *later* than 3.1.0, please check the revision history notes that come with it in case there are any changes that you need to implement to the code in heidi.t, and the code given in the remainder of this *Guide* to get things to work.

Exactly how you approach this *Getting Started Guide* may depend both on your temperament and your previous experience (either with other systems for writing Interactive Fiction or with programming in general). If you feel you want a reasonably sound grasp of the findamentals of the TADS 3 language before seeing how to write a game in it, then you'll probably want to read the remainder of this chapter. If, on the other hand, you'd rather get straight on with the business of seeing how a game is written, you might prefer to skip straight to Chapter Two and only return to the last two sections of this chapter if and when you feel the need to clarify the programming concepts employed in the course of this guide. If you'd prefer a compromise solution, you could read the "Programming Prolegomena" section below, but leave the final "Further Programming Concepts and Constructs" section till later. But whatever you do, please try to master the basics fully before trying to do something more complex; some new users of TADS 3 become frustrated because they try to run before they have learned how to walk.

Finally, do feel free to experiment as you work your way through the material that follows; you will almost certainly learn more if you try adapting the example code given to try things out than if you simply copy it all with your brain in neutral. It may often, however, be a good idea to clean out your experimental code before going on to the next step in this *Guide*, just in case your bright ideas clash with mine, and if you find your experiments are leading to frustration and confusion you may want to postpone them for a while.

2. *Creating your First TADS 3 Project*

a. *Installing the TADS 3 Author's Kit*

If you're using Windows, there's almost nothing to this - just download the TADS 3 Author's Kit, which consists of a single .EXE file that installs everything. Open the installer executable (by double-clicking on it from wherever you downloaded it to), and step through the install screens. Everything should be self-explanatory. When the install is finished, you're all set.

For Macintosh and Unix systems, refer to the README file that comes with your system's download package for instructions (for Macintosh see also section b.ii below).

b. *Creating the new project*

i) Creating a project with Workbench

If you're using Windows, run TADS 3 Workbench (by selecting it from the "Start" menu group you selected during the installation process).

- By default, Workbench will show you a "welcome" dialog asking you if you want to open an existing game or create a new one. Click on the button for creating a new game.
- If you've turned off the "welcome" dialog, then select "New Project" from the Workbench "File" menu.

In either case, this will display the New Project Wizard. Just step through the wizard screens to tell Workbench the name and location for your new project files. Workbench will automatically create all of the necessary files for your project, and it'll even compile it for you right away.

The steps you'd typically follow once the wizard is launched would be:

1) Click the 'Browse' button on the first page of the Wizard.
2) Use the file dialog that appears to create a new folder (e.g. 'MyNewGame') under your TADS 3 folder.
3) Navigate to the new directory you have just created and enter a filename for your new game (e.g. 'MyNewGame') into the File Name field of the dialog, and then click the 'Save' button.
4) Click the 'Next' button on the wizard.
5) Click the 'Next' button again. On the next page of the wizard select the 'Advanced' radio button (for the purposes of this *Guide* you don't want the 'Introductory' option).
6) Click the 'Next' button again. On the next page of the wizard leave the 'Standard' radio button selected (the WebUI is a topic beyond the scope of this *Guide*).
7) Fill in the four fields on the next page of the Wizard. Under 'Story Title' put the full name of your game ('My New Game' or whatever you want to call it; later on in this *Guide* it will 'The Further Adventures of Heidi' for example). Then fill in the next

two fields with your own name and email address. The final field can be used to give a brief description of the game (e.g. ‘This is simply a tutorial game I’m using to learn TADS 3 with’ or ‘Heidi has further adventures in the forest and makes some new friends’).

8) Click ‘Next’ and then click ‘Finish’.
9) Wait until TADS 3 Workbench has finished created and compiling the new skeleton game (you should see a message saying ‘Build successfully completed' followed by the date and time). In the left-hand pane of Workbench (headed ‘Project’) look for the section (near the top) that says ‘Source Files’ and double-click on the icon representing the file you asked the Wizard to create at Step 3 above (e.g. ‘MyNewGame.t’); it should be the third one down. You will then see your new game source file open in the Workbench editing window.
10) To compile the project again when you’ve made changes to it, just press the F7 key. (You can also select the "Compile for Debugging" command on the "Build" menu, or click the equivalent toolbar button.)

ii) Creating a project manually (for non-Windows Users)

If you're *not* using Workbench (which at this stage should only be because you're not using Windows), you'll have to create your project files manually. Fortunately, this isn't very hard - you just need to create two files and one subdirectory.

Jim Aikin suggests the following steps for setting up TADS 3 and creating a project on a Macintosh (these should also work for other non-Windows systems with a little adaptation):

1) Download FrobTADS, double-click on the .dmg file, and run the installer.

2) Create a directory to hold your projects, and a subdirectory within it to hold your first project. For example, in Documents, create TADS. In TADS, create a MyGame folder.

3) In the folder for your first project, create a folder called obj. This will hold the object files created by the compiler while it's running. You won't need to be concerned about anything in this folder; it will take care of itself.

4) Using a text editor, create a .t3m file. For convenience, give the .t3m file the same name as the project, perhaps MyGame.t3m. Copy the following text into your new .t3m file and save the file to the project folder:

```
-DLANGUAGE=en_us
-DMESSAGESTYLE=neu
-Fy obj -Fo obj
-o MyGame.t3
-lib system
-lib adv3/adv3
-source MyGame
```

Replace "MyGame" in the code above with the name of your actual game, if it's different.

5) 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.

6) Create a starter game file, again as a text file, and save it to the MyGame directory. Your starter game should look more or less like this:

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

gameMain: GameMainDef
  initialPlayerChar = me
;

versionInfo: GameID
  name = 'My First Game'
  byline = 'by Bob Author'
  authorEmail = 'Bob Author <bob@myisp.com>'
  desc = 'This is an example of how to start a new game project. '
  version = '1'
  IFID = 'b8563851-6257-77c3-04ee-278ceaeb48ac'
;

firstRoom: Room 'Starting Room'
  "This is the boring starting room."
;

+me: Actor
;
```

Fill in those quoted parts under the line reading "versionInfo:GameID" with your own information. Everything should be self-explanatory, except that last line that starts "IFID =". That long, random-looking string of letters and numbers is exactly what it appears to be - a long, random string of letters and numbers. Well, almost: it's actually composed of random "hexadecimal", or base-16, digits, i.e. 0 to 9 plus A to F. The purpose of this random number is to serve as a unique identifier for your game when you upload it to the IF Archive. The format is important, but the individual digits should simply be chosen randomly. For your convenience, tads.org provides an on-line IFID generator at http://www.tads.org/ifidgen/ifidgen.

7) 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/MyGame' and then hit Return.

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

```
t3make -d -f MyGame
```

If all goes well, you should see a string of messages in the Terminal window, and a new file (MyGame.t3) will appear in the MyGame 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 MyGame.t3' and hitting Return.

9) Instead of typing the t3make command every time you want to compile your game, you can create a .command file in your project folder and then double-click this file. Double-clicking the .command file will launch Terminal and pass the text in the .command file to Terminal.

However, when you try this, the Macintosh is quite likely to object that you don't have permission to execute the .command file. There seems to be no way to fix this using the Info box (which is opened using Cmd-I). You'll have to do it from the Terminal. Navigate, as before, to the directory where your .command file is located, and type this into the Terminal:

```
chmod +x MyGame.command
```

Again, substitute the name of your actual .command file. The chmod instruction with a +x flag will make the .command file executable. Now you can double-click it to compile your game, but only if Terminal is already logged into the game's directory. If no Terminal window is open, that won't be the case. To remedy this problem, add the cd line to the beginning of the .command file, so that the .command file looks something like this (substituting the name of your directory and game file):

```
cd Documents/TADS/MyGame
t3make -d -f MyGame
```

c. *Running your game*

If you're running Workbench, once again, this is easy - press the F5 key (or select the "Go" command on the "Debug" menu, or click the equivalent toolbar button).

If you're not running Workbench, at your system command prompt, type

```
t3run mygame
```

But you should check the README file that came with your system's download package - the program name might not be the same everywhere.

If you want more advanced instructions, or you'd like a fuller explanation of what everything means, please read the article on ‘Creating Your First TADS 3 Project’ in the *TADS 3 Technical Manual*. When you come to create a larger project you might want to split it over several source files, which is explained in the article on ‘Separate Compilation’ in the *Technical Manual*, but this is not something you need worry about for the purposes of this Guide.

3. *Programming Prolegomena*

Many readers may prefer to skip this section altogether and dive straight into the more interesting business of writing a game. But if you are completely new to programming in TADS (or TADS 3) you may appreciate a brief introduction to some of the basic ground rules. This section makes no attempt to give a comprehensive or systematic account of the TADS 3 language, but simply introduces some of the things you will be meeting in this *Getting Started* Guide.

a. *Overview of Basic Concepts*

Writing a game in TADS 3 requires two different styles of programming: *declarative* and *procedural*. *Declarative* programming is largely a matter of defining *objects* and setting their *properties* (see below). Setting the properties of objects means giving them *values*; a *value* may typically be a number, a string (i.e. a piece of text) or another object. Since adv3, the library that comes with TADS 3, is so rich, you can achieve a great deal in TADS 3 with declarative programming alone.

Procedural programming involves writing a sequence of *statements*. Each *statement* is an instruction that you want your game to carry out. Statements may typically assign a value to a *variable* or property, or call a *function* or *method*. A *variable* is a kind of temporary store for a value; a *property* can act as a more permanent store.

With one or two exceptions we needn't worry about here, *statements* can appear only in *functions* and *methods*; there needs to be some context in which they are executed. Similarly, *variables* can only be used in *functions* and *methods*; all TADS 3 variables are thus *local* variables (see further below).

A *function* is a kind of wrapper for a group of related statements you want to be executed together. An individual function is usually designed to carry out one specific task (although it may be a highly complex task involving many individual steps). The process of telling TADS 3 that we want a function to carry out its task is known as *calling* or *invoking* the function (the two terms are synonymous).

A *method* is similar to a function, but is associated with a particular *object*. A function can be invoked (i.e. called) simply using its name (e.g. the statement `foo()` will invoke the function named `foo`), whereas invoking a method generally requires specifying the name of the object to which it belongs as well (e.g. `foo.bar()` would invoke the bar method of the `foo` object). The exception is when a method is invoked from another method of the same object.

b. *Objects*

Broadly speaking, most programming in TADS consists of defining *objects* (although you may also find yourself defining classes, functions, and one or two other things, but we'll leave those to one side for the moment). An object may be an object in the physical sense of something that appears in your game world, such as a spade, a cottage, or a shopkeeper, but it may also be a more abstract construct designed to do some job or other in your code. Examples of some of abstract objects we shall be encountering include ActorStates that help describe how an actor behaves under particular circumstances, and TopicEntries that define how an actor responds to various questions.

Objects generally belong in some form of *containment hierarchy*. For physical objects this usually represents the notional containment relationships in your game world. At the top of the hierarchy are the rooms (locations) that make up the map of your world. Each individual room may contain a number of objects, such as tables, chairs, rocks, boxes and the like, as well as actors such as the player character (PC) and non-player characters (NPCs). These in turn may 'contain' further objects (and so on). For example, if there is coin inside one of the boxes, the coin is contained by the box, just as the box is contained by the room. 'Containment' is, however, a slightly more general relation than this example might suggest. For example, if a pen is sitting on the table, then the table is considered to be the pen's container. Anything held (or worn) by an actor is considered to be contained by the actor. So, for example, if the PC picks up one of the rocks, that rock's container changes from the room to the PC. If the PC then puts one of the boxes on the table, the box is now 'contained' by the table instead of directly by the room (although it remains indirectly contained by the room). At this point the coin is contained by the box, but is also 'in' the table and the room. In TADS 3 the immediately container of an object is always specified in its `location` property.

Containment may also be used to relate abstract objects. For example, menu items may be contained in a menu, or an actor may 'contain' abstract objects such as ActorStates and TopicEntries (these will be explained in due course) as well as physical objects being carried around by the actor.

Typically an object definition begins with the name of an object, followed by a colon, followed by a class list, followed by a list of its `properties` and `methods`:

```
myObj : Thing
    name = 'boring object'
    changeName
    {
       name = 'even more boring object';
    }
;
```

In this definition `name` is a *property* of `myObj`, `changeName` is a *method* and `Thing` is the *class* (or *superclass* or *base class*) of the object. The functional difference between a *property* and a *method* is that properties hold values while methods contain code: a list of one or more statements that do something when the method is invoked. The syntactical difference is that the name of a property is separated from its value by an equals sign (=) while that of a method is not, the statements that make up the method being enclosed in braces { }.

A further point of syntax to note is the use of the semicolon. This is used (a) to terminate the object definition, and (b) to terminate statements. It is *not* used to terminate property definitions (a very, very easy mistake to make). Although they look very similar, the line `name = 'boring object'` is a property definition that means "define a `name` property on `myObj` and set its initial value to 'boring object'", while the statement within the `changeName` method, i.e. `name = 'even more boring object';` is an assignment statement that means "change the value of the already existing value of the `name` property to 'even more boring object'."[1]

[1] TADS 2 users should note that the TADS 3 assignment operator is always = and not :=; TADS 3 follows C conventions and not Pascal conventions throughout.

Note that you could use braces instead of a terminating semicolon to define the extent of the object definition; the foregoing object definition could then have been written:

```
myObj : Thing
{
    name = 'boring object'
    changeName
    {
       name = 'even more boring object';
    }
}
```

Which you use is up to you, but this *Guide* will use the terminating semicolon.

c. *Assignment Statements*

An assignment statement is probably one of the most common kinds of statement that you will come across in TADS 3 programming. It always takes the form:

```
lvalue = expression;
```

Where `lvalue` can be either an object property or a variable (which we'll talk about in just a bit). An expression can be as simple as a constant value or the name of another variable, a function call or method name (assuming the function or method returns a suitable value), or a more complex expression involving a number of the foregoing elements joined together with `operators`, for example:

```
myName = 'my '  + name;
```

As a statement this would assign the value 'my boring object' to the variable `myName` (assuming that `name` started off by holding the value 'boring object'). Note that an expression can also be used as the value of a *property* (in which case it should be enclosed in parentheses), so that if we made `myName` a *property* of myObj, we could definine it thus:

```
myObj : Thing
    name = 'boring object'
    changeName
    {
       name = 'even more boring object.';
    }
    myName = ('my ' + name)
;
```

This definition would mean that `myName` contained 'my boring object' until the `changeName` method was invoked, and would contain 'my even more boring object' afterwards (we'll talk about invoking methods presently). In fact, it is, except for its appearance, *exactly* the same as writing:

```
myName  { return 'my ' + name; }
```

When it is used with (single-quoted) strings, + is thus a concatenation operator. With numbers it does what you would expect, i.e. add them together, e.g.:

```
myNumber = 3 + 4;
```

Would assign the number 7 to `myNumber`. All the numbers we'll be dealing with in this Guide will be *integers* (i.e. whole numbers); TADS 3 does possess a `BigNumber` class that allows you to work with real numbers (i.e. numbers including a fractional part, such as 3.14159[2]), but most Interactive Fiction can get by quite happily with standard integer arithmetic.

Other common arithmetic operators include -, * and / (subtract, multiply and divide) which do much what you would expect (note that the division is integer division, so that `myNumber = 3 / 4` would set `myNumber` to zero, while `myNumber = 10 / 4` would set it to 2). Less obvious but almost just as common and useful are the various shortcut operators that provide a more concise way of coding common operations. There are several of these, but the only ones we need deal with here are += -= ++ and --. It is quite common in programming to want to add or subtract a number from the current value of a variable or property and store the result in the same variable or property, e.g.:

```
myNumber = myNumber + 4;
myNumber = myNumber - 2;
```

If `myNumber` started out at 6, then after the first line was executed, myNumber would be changed to 10, and after the second line was executed, it would be changed to 8. This could be written more succinctly as:

```
myNumber += 4;
myNumber -= 2;
```

This may look a litle strange at first, but it's a highly convenient feature once you get the hang of it. Another one is the use of ++ or -- to increase or reduce a property or variable by one. Thus intead of writing `myNumber = myNumber + 1` or even `myNumber+=1` one could write simply `myNumber++`; likewise one could use `myNumber--` in place of `myNumber = myNumber - 1`.[3]

In these examples, `myNumber` could be either a property or a variable. In TADS 3 programming properties tend to be used for semi-permanent storage of information you need to be available to the whole program, while variables are local in scope and temporary in duration, used, for example, to hold the results of some intermediate calculation (there are some library defined quanties of the form `gWhatsit` that look like global variables, but these are simply shorthand ways of referring to some

[2] Mathematically speaking *real numbers* should be contrasted with *imaginary* or *complex* numbers, such as 3 + 2.4i, but that's a complication we'll ignore here.

[3] For a full account of these operators you need to be aware of the difference between the postfix operator myNumber++ and the prefix operator ++myNumber, the difference being that in the former case myNumber is incremented *after* its value is used, and in the latter it is incremented beforehand. This would become revelant in a situation where, for example, you wrote x = 3 + myNumber++ or x = 3 + ++myNumber. Assuming myNumber started out at 2, the first example would result in x becoming 5 and myNumber becoming 3, while the second would result in myNumber becoming 3 and x becoming 6. At this introductory stage this is not worth worrying about too much, but is something you will need to master if you plan to make a lot of use of these operators.

commonly used property of a library object). Being local in *scope* means that the variable is available only to code within the same block (usually the same method or function) as that in which the variable is defined; being temporary in *duration* means (to a first approximation) that the variable only retains its value for that particular invocation of the function or method. A variable must be declared with the keyword `local` in the block in which it appears, and may optionally be initialized in the same statement in which it is initialized, e.g.:

```
local x;
local numberOfCabbageEaters = 12;
```

d. *Referring to Methods and Properties*

Variables, and indeed statements, are generally used within object methods and global functions. But how are the functions and methods used in turn? Often the library will expect a method to be defined on an object you create and will invoke (call) it under the appropriate circumstances; moreover, you can often use a method in place of a property when you want to do something more complex than you can do with a property; then, when the library tries to (say) display the value of the `name` property it may quite happily use the value returned by the `name` method instead. If you've defined a method `myMethod` on an object `myObj` you can invoke it from anywhere in your code by writing the statement:

```
myObj.myMethod;
```

or

```
myObj.myMethod();
```

Similarly, you can reference the value of the `myProperty` property of `myObj` with `myObj.myProperty`. Note the use of the dot (.) notation here, since you will be using it a lot.

In TADS 2 (or Inform 6), if you wanted to reference `myObj.myMethod()` or `myObj.myProperty` from another property or method of `myObj` you would typically write `self.myMethod()` or `self.myProperty(),` where `self` is a special keyword meaning "the current object". There are still situations where you may need to use the `self` keyword in TADS 3 but this is no longer one of them; instead, in this situation, you could write simply, `myMethod()` or `myProperty`. To make this clearer, we'll give an example:

```
myObj : Thing
    name = 'boring object'
    changeName
    {
        name = 'very boring object';
    }
    myName = ('my ' + name)
;

myOtherObject : Thing
   name = 'exciting object'
   describeName
   {
       local dName = 'This is an ' + name + ', unlike ';
```

```
        myObj.changeName;
        dName +=  myObj.myName;
        say(dName);
        return dName;
    }
;
```

In this example, a call to `myOtherObj.describeName` should result in the display of the message "This is an exciting object, unlike my very boring object"; moreover, if you wrote a statement such as `msg = myOtherObj.describeName`, not only would "This is an exciting object, unlike my very boring object" be displayed, but the string 'This is an exciting object, unlike my very boring object' would be stored in the variable `msg`.[4] This comes about because the last statement of `describeName` tells the method to return a value (in this case the value of the local variable `dName`), and this value will be treated as the value of the method if it is used in an expression.

e. *Functions and Methods*

Functions may return values in similar ways. The purpose of using a function is typically to perform an often-used calculation that is not related to any particular object, e.g.:

```
function salesTax(salesValue, taxPercent)
{
    return (salesValue * taxPercent)/100;
}
```

. The `function` keyword used here is optional but perhaps makes the code clearer, although it is more usual to omit it in TADS 3 code. Note that in this example, unless we're using the `BigNumber` class, `salesValue` and `taxPercent` must both be integers (e.g. 120 meaning, say, 120 pence or 120 cents, and 15 meaning 15%). More to the point, note that `salesValue` and `taxPercent` are the two formal parameters of this function, which means that they're placeholders for whatever values we want to pass to the function when we call it. So, for example, if from somewhere in the program we called `taxPennies = salesTax(120, 15);` `taxPennies` would be assigned the value 18. Methods may also take parameters, so for example we could define:

```
myObj : Thing
   baseName = 'object'
   myName (qualifier)
   {
      return 'my ' + qualifier + ' ' + baseName;
   }
;
```

Note the use of extra string spaces so that `myObj.myName('boring')` returns 'my boring object' rather than 'myboringobject'. Note also that we can also define a method (or function) that takes no arguments by using an empty argument list thus: (). So, for example, we could have defined:

[4] For this distinction between single and double quoted strings, see p. 51 below.

```
myObj : Thing
    name = 'boring object'
    changeName()
    {
        name = 'very boring object';
    }
    myName = ('my ' + name)
;
```

And it would have meant precisely the same as the earlier definition without the empty () after `changeName`. Which you use is entirely up to you.

f. *Conditions - If Statements*

Often one will want to use methods and functions to perform something a bit more complex than we've shown here. One of the basic requirements of any programming language is to be able to test for conditions and act according to the results. For example, we might want `myObj` to declare itself as either a boring object or exciting object on the basis of a property used as a flag:

```
myObj : Thing
   name
   {
       if(exciting)
           return 'exciting object';
       else
           return 'boring object';
   }
   exciting = nil
   myName = ('my ' + name)
;
```

The new construction introduced here means "if the condition in parentheses following the keyword 'if' is true, carry out the statement on the following line, otherwise carry out the statement following the 'else' keyword". TADS 3 defines two special values, `true` and `nil` which mean true and false (N.B., it's *very* easy, especially if you're used to using another language, to type 'false' when you mean 'nil'; TADS 3 uses `nil` since it has other uses beyond Boolean false[5]). Since the property `exciting` contains `nil` `myObj.name` will return 'boring object'; if `exciting` were later changed to `true` (or to any non-zero number), `myObj.name` would then return 'exciting object'.

The condition in an `if` statement can be much more elaborate than the name of a property that evaluates to `nil` or `true`. For example, suppose that instead of a boolean (`nil` or `true`) `exciting` property we defined a numeric `excitement` property, with the rule that the object only becomes exciting if its `excitement` property exceeds 10. We should then have written the test as `if(excitement > 10)`. Alternatively, we might have decided that the object value was only exciting if its excitement value was exactly 123, in which case the condition would be written `if(excitement == 123)`.

Note that this test for equality uses a *double equals sign* (==), and must be written this way if this is what you mean. It's very easy to write something like

[5] For example, a property or method returns a nil value if is undefined, game objects can be moved into nil to move them out of the game world, and in some situations a value of nil can be equivalent to an empty string.

`if(excitement = 123)` by mistake, in which case the compiler will give you a warning, because it almost certainly isn't what you meant.[6]

You may also want to combine tests using the logical operators *and*, *or* and *not*, which in TADS 3 are defined with `&&`, `||` and `!` respectively. For example if we have defined a `boring` property on `myObj`, we might have wanted the exciting test to be:

```
if((!boring && excitement > 12) || excitement == 123)
```

This would mean, if excitement is equal to 123 or if it's greater than 12 and boring is not true. Note the use of grouping parentheses to resolve any potential ambiguities in the order in which these conditions are evaluated.

There is no need to use the else clause at all, if you don't need it. But what happens if you need more than one statement to be executed if something is true, and/or a whole set of statements to be performed otherwise? In this case, we'd use braces {} to group the statements, for example:

```
if((!boring && excitement > 12) || excitement == 123)
    {
        myIndefiniteArticle = 'an';
        return 'exciting object';
    }
else
    {
        myIndefiniteArticle = 'a';
        return 'boring object';
    }
```

It’s quite common to want to assign one value to something if a condition holds, and another otherwise, for example:

```
if(length > 5)
    size = 'big';
else
    size = 'small';
```

This is kind of thing is so common that TADS 3 provides a short-cut way of doing it. Instead of writing the above, you could write simply:

```
size = (length > 5) ? 'big' : 'small';
```

More generally this *ternary operator* works as follows:

```
cond ? true-value  : false-value
```

If `cond` is true this evaluates to `true-value`, otherwise it evaluates to `false-value`.

[6] Once again TADS 2 users should note the difference from TADS 2 here; this is another case where there is no longer the option to use Pascal-style syntax.

g. *The Switch Statement*

It is possible to nest `if… else…` statements to any required depth, so that one could, for example, have the following:

```
if(excitement == 0)
   name = 'very boring object';
else if (excitement == 1)
   name = 'boring object';
else if (excitement == 2)
   name = 'moderately boring object';
else if (excitement < 5)
   name = 'vaguely boring object'
else if(excitement < 10)
   name = 'not too boring object';
else
   name = 'exciting object';
```

But the trouble with this is that it can quickly become confusing to keep track of which `else` is meant to match which `if` (this can be alleviated by using braces to group the code the way you want, though that can lead to messy-looking and verbose code). In some cases this may be the only way to achieve the effect you want, but in this particular case, where we are simply testing the value of a single variable, it is often easier to use a *switch* statement; in this case the equivalent switch statement would be:

```
switch(excitement)
{
   case 0: name = 'very boring object'; break;
   case 1: name = 'boring object'; break;
   case 2: name = 'moderately boring object'; break;
   case 3:
   case 4: name = 'vaguely boring object'; break;
   case 5:
   case 6:
   case 7:
   case 8:
   case 9: name = 'not too boring object'; break;
   default : name = 'exciting object';
}
```

Note the use of the `break;` statements to stop the test 'falling through' to other matches. Since we want the test to fall through if `excitement` is 3, 5, 6, 7 or 8 we do not define a `break` statement for those cases. So, for example, if `excitement` is 6 the `switch` statement will execute the statements for all the cases following `case 6` until it encounters a `break`; this has the desired effect of setting name to 'not too boring object'. The `default` case defines what happens if none of the preceding cases is matched.

The `switch()` statement is not restricted to matching numbers, it can also match (single-quoted) strings, objects, lists, Boolean values (true or nil) or enumerators (which we'll meet again below). Again, the `case` value need not be expressed as a constant of one of these types, so long as it is an expression that evaluates to a constant value of one of these types.

h. *Properties Containing Objects and Lists*

This brings us to the final introductory point: so far our examples of properties and variables have all been of ones that contain either numbers, strings, or Boolean values (true or nil); but properties and variables can also contain other data types such as objects, lists, enumerators and function pointers, and although we shall be meeting few enumerators and function pointers in what follows, properties containing objects and lists will be rather more common. The concept of a property or variable containing an object is really no more complicated than that of having them refer to strings or numbers. For example if we had two objects, `myObj1` and `myObj2`, we could, say, use the assignment statement `obj = myObj2` and then use obj to refer to `myObj2`. This may seem a bit pointless at first, but it could be useful if we didn't know in advance which object obj was going to be, and we wanted to write general code that could work equally well with a number of objects. To take a trivial example, suppose we wrote the following function:

```
function showName(obj)
{
   say(obj.name);
}
```

This definition would then allow us to call `showName(myObj1)` to display `myObj1.name`, `showName(myObj2)` to display `myObj2.name` and so on. This example is so trivial that it may still seem pointless, but even in a slightly more complex case the value may start to become apparent:

```
function talkAbout(obj)
{
    local msg = 'My ';
    msg += obj.name;
    msg += ' is really very ';
    if(obj.excitement<10)
       msg += 'dull.';
    else
      msg += 'interesting.';
    say(msg);
}
```

Perhaps an even more common use of assigning objects to properties is where other objects need to keep track of them. For example, if I have an object (say 'ball') inside another object ('bag'), then the `location` property of the ball can keep track of where the ball is by being set to the bag object. If the ball is then moved to the tennis court the `location` property of the ball object could be set to the `tennisCourt` object to keep track of it.

At first sight, it may seem that doing it the other way round wouldn't work so well, since, say, using `bag.contents` to keep track of what's in the bag would only allow one object to be in the bag at the time. In fact this is an example of where one would use a `list` value. A list is basically a list of items (of any of the valid types, including other lists) enclosed in square brackets and separated by commas, e.g.:

```
bag.contents = [ball, coin, banana, horseshoe]
```

To find out whether something's in a list one can use its `indexOf` method; e.g. `bag.contents.indexOf(ball)` would be 1; `bag.contents.indexOf(banana)` would be 3, and `bag.contents.indexOf(elixirOfLife)` would be nil.

i. *Nested Objects*

The previous section only scratches the surface of TADS 3 lists; to find out more, look up lists in the *System Manual* that comes with the TADS 3 Author's Kit. We'll conclude with a rather different kind of list to illustrate one last point, the use of nested objects in TADS 3.

Suppose we have a ball that appears to change colour randomly when we look at it. We might define it like this:

```
ball: Thing 'ball' 'ball'
   "When you look at it, it looks <<colour>>. "
    colour  { return colourList.getNextValue(); }
;

colourList: ShuffledList
   valueList = ['red', 'green', 'blue', 'violet', 'white',
     'black', 'orange', 'indigo']
;
```

The purpose of a `ShuffledList` is to return one of its values randomly, without repeating a value until it has used them all. It's a bit like shuffling a pack of cards, then taking one in turn until all have been used, then reshuffling the pack and starting again. But in order to function this way a `ShuffledList` needs to be a separate object, not only with a list of values (its `valueList` property) but also a method (`getNextValue`) that returns the next shuffled value. In order to define the varicoloured ball object, therefore, we also need to define a separate `colourList` object. While this is far from catastrophic, it can be a little inconvenient, since code that helps to define the behaviour of one object is spread into another; the two objects might in time get separated in your code, or the presence of the second object might mess up the containment hierarchy in some way. This is where a nested object could come in handy.

As an intermediary step, note that a property can contain a reference to an object; for example, we could have written:

```
ball: Thing 'ball' 'ball'
   "When you look at it, it looks <<colour>>. "
    colour  { return colourList.getNextValue(); }
    colourList = colourListObj
;

colourListObj: ShuffledList
   valueList = ['red', 'green', 'blue', 'violet', 'white',
     'black', 'orange', 'indigo']
;
```

And this would work just the same (although it appears a little more verbose). The `colour` method now refers to the `colourList` property which in turn refers to the `colourListObj` object. The way we could make this more compact is to turn the `colourListObj` object into an anonymous object defined directly on the `colourList` property:

```
ball: Thing 'ball' 'ball'
   "When you look at it, it looks <<colour>>. "
    colour  { return colourList.getNextValue(); }
    colourList: ShuffledList
    {
      valueList = ['red', 'green', 'blue', 'violet', 'white',
        'black', 'orange', 'indigo']
    }
;
```

Not only is this more concise, but it has the advantage of keeping all the code together in one object. The `ShuffledList` has now become an *anonymous nested object*. All nested objects are anonymous, because they have no name: `colourList` is not the name of the `ShuffledList` object here, it's the name of a property of `ball`; the `ShuffledList` can nevertheless be referred to as `ball.colourList`, since it is the value of `ball`'s `colourList` property. Note that while an ordinary object definition may either be terminated with a semicolon or enclosed in braces, the braces (`{}`) form must *always* be used with a nested object, as here.

This may seem a bit strange and convoluted at first, but you'll find the use of anonymous nested objects is a powerful and common feature of TADS 3 programming, so it will be as well to become familiar with it.

4. *Further Programming Concepts and Constructs*

We have only scratched the surface of the TADS 3 language here, but further details are available in the *System Manual*, and in the meantime we have covered most of the basic features of the language that you need to follow this *Guide*. A few more will be introduced as we go along. There are, however, a few more fairly basic TADS 3 programming concepts we haven't covered yet, and as you'll probably need them sooner rather than later they're introduced here. It is not necessary to master these in order to use the rest of this *Guide*, however, so you may prefer to skip this section for now and get on with the more interesting business of discovering how to construct your first TADS 3 game, returning to this section later on if you need to.

a. *Comments, Identifiers and Scope*

i) Comments

Outside of a quoted string, two consecutive slashes, `//`, indicate that the rest of the line is a comment. Everything up to the next newline is ignored. Alternatively, C-style comments can be used; these start with `/*` and end with `*/`; this type of comment can span multiple lines.

Examples:

```
// This line is a comment.

/*
This is a comment
which goes across
several lines.
*/
```

ii) Identifiers

Identifiers must start with a letter (upper or lower case), and may contain letters, numbers, dollar signs, and underscores. Identifiers can be up to 39 characters long. Upper and lower case letters are distinct (so that, for example, cloakroom, Cloakroom and CloakRoom are three different identifiers).

iii) Scope of Identifiers and Local Variables

All objects and functions are named by global identifiers. No identifier may be used to identify different things; that is, no two objects can have the same name, an identifier naming a function can't also be used for an object, and so forth.

Property names are also global identifiers. A name used for a property can't be used for a function or object, or vice versa. However, unlike functions and objects, the same property name can be used in many different objects. Since a property name is never used alone, but always in conjunction with an object, the TADS compiler is able to determine which object's property is being referenced even if the same name is used in many objects.

Function arguments and local variables are visible only in the function in which they appear. It is permissible to re-use a global identifier as a function argument or local variable, in which case the variable supersedes the global meaning within the function. However, this is discouraged, as it can be a bit confusing.

Local variables in functions and methods must be declared with the keyword `local` before they are used. Local variable declarations can appear anywhere within a code block. A local variable definition that appears in the middle of a code block creates a variable that is in scope from that point in the code block to the closing brace of the code block. (TADS 2 only allowed local variable declarations at the start of a code block.)

You can define local variables for the current code block. This is done with a statement such as this:

```
  local identifier-list ;
```

The *identifier-list* has the form:

```
  identifier [ initializer ] [, identifier-list ]
```

An *initializer*, which is optional, has the form:

```
= expression
```

where the *expression* is any valid expression, which can contain arguments to the function or method, as well as any local variables defined prior to the local variable being initialized with the expression. The expression is evaluated, and the resulting value is assigned to the local variable prior to evaluating the next initializer, if any, and prior to executing the first statement after the `local` declaration. Local variables with initializers and local variables without initializers can be freely intermixed in a single statement; any local variables without initializers are automatically set to `nil` by the run-time system.

The identifiers defined in this fashion are visible only inside the function or method in which the `local` statement appears (actually, the situation can be slightly more complex than this when anonymous functions are involved - see the section on Anonymous Functions in the *System Manual* for the full story). Furthermore, the local statement supersedes any global meaning of the identifiers within the function or method.

An example of declaring local variables, using multiple `local` statements, and using initializers is below.

```
f(a, b)
{
  local i, j;                  /* no initializers */
  local k = 1, m, n = 2;       /* some with initializers, some without */
  local q = 5*k, r = m + q;    /* OK to use q after it's initialized */
  for (i = 1 ; i < q ; i++)
  {
    local x, y;                /* locals can be declared in any block */
    say(i);
   }
 }
```

b. *Loops*

A common programming requirement – and one that can turn up quite frequently in TADS programming – is the need to repeat a statement or set of statements a number of times. This may either be a fixed number of times or, more commonly, a number of times determined by some condition, such as the number of objects in a set we wish to examine. For example at the start of the game we might want to go through every object in the game and ensure that it is has been added to the `contents` property of its immediately container (the library in fact does this for us). It would be tedious indeed to have to write code to do this on every single object that might be affected; it's far better to write a set of statements once and have that same set of statements executed for every relevant object in our game. A programming construct that accomplishes this sort of task is traditionally called a *loop*, and the TADS 3 language provides four types of loop: while, do-while, for and foreach. It also contains a number of statements to help control how loops function.

i) While

The `while` statement defines a loop: set of statements that is executed repeatedly as long as a certain condition is true.

```
while ( expression ) statement
```

As with the `if` statement, the *statement* may be a single statement or a set of statements enclosed in braces. The *expression* should evaluate to a number (in which case 0 is false and anything else is true), or a truth value (`true` or `nil`).

The *expression* is evaluated *before* the first time through the loop; if the *expression* is false at that time, the statement or statements in the loop are skipped. Otherwise, the statement or statements are executed once, and the *expression* is evaluated again; if the *expression* is still true, the loop executes one more time and the cycle is repeated. Once the *expression* is false, execution resumes at the next statement after the loop.

For example, suppose we had a custom class called `Book` and we wanted to loop through every `Book` in our game setting its `hasBeenRead` property to `nil` unless its `author` property refers to the Player Character. We could write:

```
local obj = firstObj(Book);
while(obj != nil)
{
    if(obj.author==gPlayerChar)
       obj.hasBeenRead = true;
    else
       obj.hasBeenRead = nil;
    obj = nextObj(obj, Book);
}
```

ii) Do-While

The do-while statement defines a slightly different type of loop from the `while` statement. This type of loop also executes until a controlling expression becomes false (0 or `nil`), but evaluates the controlling expression *after* each iteration of the loop. This ensures that the loop is executed at least once, since the expression isn't tested for the first time until after the first iteration of the loop.

The general form of this statement is:

```
do statement while ( expression );
```

The statement may again be a single statement or a set of statements enclosed in braces. The expression should again evaluate either to a number (in which case 0 is false and anything else is true), or a truth value (`true` or `nil`).

For example, to calculate factorial n:

```
factorial(n)
{
    local x = 1, res = 1;
    do
       {
           res = res * x;
           x++;
       }
    while (x <= n);
    return res;
}
```

iii) For

The `for` statement defines a very powerful and general type of loop. You can always use `while` to construct any loop you can construct with `for`, but the `for` statement is often a much more compact and readable notation for the same effect.

The general form of this statement is:

```
for ( init-expr; cond-expr; reinit-expr ) statement
```

As with other looping constructs, the *statement* can be either a single statement, or a block of statements enclosed in braces.

The first expression, *init-expr*, is the "initialization expression." This expression is evaluated once, before the first iteration of the loop. It is used to initialize the variables involved in the loop.

The second expression, *cond-expr*, is the condition of the loop. It serves the same purpose as the controlling expression of a `while` statement. Before each iteration of the loop, the *cond-expr* is evaluated. If the value is true the body of the loop is executed; otherwise, the loop is terminated, and execution resumes at the statement following the loop body. Note that, like the `while` statement's controlling expression, the *cond-expr* of a `for` statement is evaluated prior to the first time through the loop (but after the *init-expr* has been evaluated), so a `for` loop will execute zero times if the *cond-expr* is false prior to the first iteration.

The third expression, *reinit-expr,* is the "re-initialization expression." This expression is evaluated *after* each iteration of the loop. Its value is ignored; the only purpose of this expression is to change the loop variables as necessary for the next iteration of the loop. Usually, the re-initialization expression will increment a counter or perform some similar function.

Any or all of the three expressions may be omitted. Omitting the expression condition is equivalent to using `true` as the expression condition; hence, a loop that starts "`for ( ;; )`" will iterate forever (or until a `break` statement is executed within the loop). A `for` statement that omits the initialization and re-initialization expressions is the same as a `while` loop.

Here's an example of using a `for` statement. This function implements a simple loop that computes the sum of the elements of a list.

```
sumlist(lst)
{
  local len = length(lst), sum, i;
  for (sum = 0, i = 1 ; i <= len ; i++)
    sum += lst[i];
}
```

Note that an equivalent loop could be written with an empty loop body, by performing the summation in the re-initialization expression. We could also move the initialization of `len` within the initialization expression of the loop.

```
sumlist(lst)
{
  local len, sum, i;
  for (len = length(lst), sum = 0, i = 1 ; i <= len ;
    sum += lst[i], i++);
}
```

You can define new local variables in the initializer part of a `for` statement by using the `local` keyword in the initializer. For example:

```
for (i = 1, local j = 3, local k = 4, l = 5 ; i < 5 ; ++i) // ...
```

This declares two new local variables, `j` and `k`, and uses the existing variables `i` and `l`. Note that `l` is *not* a new local, even though it comes after the `local k` definition, because each `local` keyword in a `for` initializer defines only one variable. Note also that an initial value assignment is required for each new local declared.

The new locals declared in a `for` initializer are local in scope to the `for` statement and its body (this is the same rule that Java uses, although note that it differs from the (undesirable) way C++ works). The effect is exactly as though an extra open brace ("{") followed by a `local` statement for each new local appeared immediately before the `for` statement, and an extra close brace ("}") appeared immediately after the end of the body of the loop.

iv) *Foreach*

The `foreach` statement provides a convenient syntax for writing a loop over the contents of a collection, such as a List or a Vector.

The syntax of the `foreach` statement is:

```
foreach ( foreach_lvalue in expression ) body
```

The *foreach_lvalue* specifies a local variable or other "lvalue" expression which serves as the looping variable. This can be any lvalue (any expression that can be used on the left-hand side of an assignment operator), or it can be the keyword `local` followed by the name of a new local variable; if `local` is used, a new local variable is created with scope local to the `foreach` statement and its body. (Note that as of TADS 3.1.0 we can use `for` in place of `foreach` in such loops, and that there are also other varieties of `for..in` loop that we needn't worry about here.)

The *expression* is any expression that evaluates to a Collection object (for which see the *System Manual*), such as a List or Vector value.

The statement loops over the elements of the collection. For each element, the statement assigns the current element to the lvalue, then executes the body.
Here's an example that displays the elements of a list.

```
local lst = [1, 2, 3, 4, 5];
foreach (local x in lst)
   "<<x>>\n";
```

v) Break and Continue

A program can get out of a loop early using the `break` statement:

```
break;
```

This is useful for terminating a loop at a midpoint. Execution resumes at the statement immediately following the innermost loop in which the break appears.

The `break` statement also is used to exit a `switch` statement. In a `switch` statement, a `break` causes execution to resume at the statement following the closing brace of the `switch` statement.

The `continue` statement does roughly the opposite of the `break` statement; it resumes execution back at the start of the innermost loop in which it appears. The `continue` statement may be used in `for`, `while`, `foreach`, and `do-while` loops.

In a `for` loop, `continue` causes execution to resume at the re-initialization step. That is, the third expression (if present) in the `for` statement is evaluated, then the second expression (if present) is evaluated; if the second expression's value is non-nil or the second expression isn't present, execution resumes at the first statement within the statement block following the `for`, otherwise at the next statement following the block.

For example, suppose we want to loop through a player's possessions until we find one that is of class `LightSource`; we might do something like this:

```
local obj;
foreach(obj in gPlayerChar.contents)
{
   if(obj.ofKind(LightSource))
      break;
}
```

The `break` and `continue` statements can optionally specify a target label. When a label is used with one of these statements, it must refer to a statement that encloses the `break` or `continue`. In the case of `continue`, the label must refer directly to a loop statement: a `for`, `while`, or `do-while` statement. The target of a `break` may be any enclosing statement.

When a label is used with `break`, the statement transfers control to the statement immediately following the labeled statement. If the target statement is a loop, control transfers to the statement following the loop body. If the target is a compound statement (a group of statements enclosed in braces), control transfers to the next statement after the block's closing brace. Targeted `break` statements are especially useful when you want to break out of a loop from within a switch statement:

```
scanLoop:
    for (i = 1 ; i < 10 ; ++i)
    {
        switch(val[i])
        {
        case '+':
            ++sum;
            break;

        case '-':
            --sum;
            break;

        case 'eof':
            break scanLoop;
        }
    }
```

Targeted `break` statements are also useful for breaking out of nested loops:

```
matchLoop:

    for (i = 1 ; i <= val.length() ; ++i)
    {
        for (j = 1 ; j < i ; ++j)
        {
            if (val[i] == val[j])
                break matchLoop;
        }
    }
```

vi) Alternatives to Loops

It seems to be one of the best kept secrets of TADS 3 that for many purposes there's often a more compact alternative to using a loop, particular when working with a *Collection* such as List or Vector. For example the foreach loop used above to identify a LightSource held by the player could have been replaced with a single statement:

```
   local obj = gPlayerChar.contents.valWhich({x: x.ofKind(LightSource)});
```

The above statement will hardly be transparent to the novice, and this probably isn't the best place to explain it, since it involves concepts that go some way beyond the introductory. At this point it must suffice to call your attention to the possibility of this kind of construct, which can be extremely powerful once mastered. To find out more (when you feel ready), read the sections on Anonymous Functions, List and Vector in the *System Manual*.

c. *Inheritance*

TADS 3 is an object-oriented language which makes heavy use of inheritance (that is to say, the language supports inheritance and the library makes heavy use of it). At its simplest inheritance allows us to have the best of both worlds: to modify the behaviour of an existing object or class but still make use of the behaviour defined on that class. For example, suppose we defined a `Switch` class with a method that defines what happens when it's switched on and off:

```
Switch: Thing
       makeOn(stat)
       {
          isOn = stat;
          "You flip the switch << stat ? 'on' : 'off' >>. "
       }
       isOn = nil
;
```

Now suppose you wanted a `LightSwitch` class that did exactly the same as the `Switch` class, but also turn on an associated light source when turned on. You could derive this from `Switch` as a subclass, and you'd still want its `makeOn` method to do everything `Switch`'s `makeOn` method does, but you'd also want it to light the light source. It would be tedious to have to retype the whole `makeOn(stat`) method, particularly in cases where it was something rather more substantial than here; instead we can inherit it and then add our own modifications:

```
LightSwitch: Switch
       makeOn(stat)
       {
          inherited(stat);
          if(myLight != nil)
            myLight.makeLit(stat);
       }
       myLight = nil
;
```

Note that we don't have to repeat the definition of the `isOn` property, since this is already inherited from the `Switch` class. We'll now examine this mechanism in a bit more detail.

i) Inherited

A special pseudo-object called `inherited` allows you to call a method in the current `self` object's superclass. Moreover, you can use `inherited` in an expression, so any value returned by the superclass method can be determined and used by the current method. Third, you can pass arguments to the property invoked with the `inherited` pseudo-object.

You can use `inherited` in an expression anywhere that you can use `self`.

Here is an example of using `inherited`.

```
MyClass: object
  sdesc = "myclass"
  prop1(a, b)
  {
     "This is myclass's prop1.  self = << sdesc >>,
      a = << a >>, and b = << b >>.\n";
      return(123);
  }
;

myobj: MyClass
  sdesc = "myobj"
  prop1(d, e, f)
  {
      local x;
      "This is myobj's prop1.  self = << sdesc >>,
      d = << d >>, e = << e >>, and f = << f >>.\n";
      x = inherited.prop1(d, f) * 2;
      "Back in myobj's prop1.  x = << x >>\n";
  }
;
```

When you call `myobj.prop1(1, 2, 3)`, the following will be displayed:

```
This is myobj's prop1. self = myobj, d = 1, e = 2, and f = 3.
This is myclass's prop1. self = myobj, a = 1, and b = 3.
Back in myobj's prop1. x = 246.
```

Note that the `self` object that is in effect while the superclass method is being executed is the *same* as the `self` object in the calling (subclass) method. This makes `inherited` very different from calling the superclass method directly (i.e., by using the superclass object's name in place of `inherited`).

You can also specify the name of the superclass after the 'inherited' keyword; this is otherwise similar to the normal 'inherited' syntax:

```
inherited Fixture.actionDobjTake();
```

This specifies that you want the method to inherit the actionDobjTake() implementation from the `Fixture` superclass, regardless of whether TADS might normally have chosen another superclass as the overridden method. This is useful for situations involving multiple inheritance where you want more control over which of the base classes of an object should provide a particular behavior for the subclass.

If the last example had been called from within the `actionDobjTake()` method of the object in question, we could simply have written:

```
inherited Fixture();
```

It is legal to omit the property name or expression in an `inherited` or `delegated` (see below) expression. When the property name or expression is omitted, the property inherited or delegated to is implicitly the same as the current target property. For example, consider this code:

```
myObj: myClass
  myMethod(a, b)
  {
    inherited(a*2, b*2);
  }
;
```

This invokes the inherited myMethod(), as though we had instead written `inherited.myMethod(a*2, b*2)`. Because the current method is `myMethod` when the `inherited` expression is evaluated, `myMethod` is the implied property of the `inherited` expression.

ii) Multiple Inheritance

An object can inherit properties from more than one other object. This is called Multiple Inheritance. It complicates things considerably, primarily because it can be confusing to figure out exactly where an object is inheriting its properties from. In essence, the order in which you specify an object's superclasses determines the priority of inheritance if the object could inherit the same property from several of its superclasses.

```
multiObj: class1, class2, class3
;
```

Here we have defined `multiObj` to inherit properties first from `class1`, then from `class2`, then from `class3`. If all three classes define a property `prop1`, `multiObj` inherits `prop1` from `class1`, since it is specified first.

Multiple inheritance can be a very useful feature. For example, suppose you wanted to define a huge vase; it should be fixed in the room, since it is too heavy to carry, but it should also be a container. With multiple inheritance, you can define the object to be both a `Heavy` and a `Container` (which are classes defined in the standard library).

If a property is inherited from more than one of its superclasses (and is not overridden in the object's own property list), the property is inherited from the superclass that appears earliest in the list. For example, suppose you define an object like this:

```
vase: Container, Heavy;
```

If both `Container` and `Heavy` define a method named `m1`, and vase itself doesn't define an `m1` method, then `m1` is inherited from `Container`, because it appears earlier in the superclass list than `Heavy`.

There is a more complicated case that can occur. You do not need to master this in order to follow this guide, so skip this section if you find it confusing. Suppose that in the example above, both `Container` and `Heavy` have the superclass `Thing`, and that `Thing` and `Heavy` define method `m2`, and that neither `Container` nor vase define `m2`. Now, since `Container` inherits `m2` from `Thing`, it might seem that vase should inherit `m2` from `Container` and thus from `Thing`. However, this is not the case; since the `m2` defined in `Heavy` overrides the one defined in `Thing`, vase inherits the `m2` from `Heavy` rather than the one from `Thing`. Hence, the rule, fully stated, is: the inherited property in the case of multiple inheritance is that property of the earliest (leftmost) superclass in the object's superclass list that is not overridden by a subsequent superclass. An alternative way of expressing this is "The first (left-most) superclass has precedence for inheritance, so any properties or methods that it defines effectively override the same properties and methods defined in subsequent superclasses, except that an ancestor class does not override a method or property on any of its descendent classes."

Don't worry if this is less than crystal-clear at the moment; simply think of it as something you may need come back to. In the meantime bear in mind two simple consequences: (1) it may not always be immediately obvious (in a situation of multiple inheritance) what the keyword `inherited` will inherit from; and (2) the order of classes in an object definition can be important (e.g `myDoor: Lockable, Door` works properly while `myDoor: Door, Lockable` doesn't).

iii) Replace and Modify

Most game authors sooner or later find that, when writing a substantial game, they need to modify the standard library behaviour at a number of points. While it would in principle be possible to modify the library files, this would create a problem when a new version of TADS is released, because you must either continue to use the old version of `adv3`, which means that any bug fixes or enhancements in the new version are not available, or take the time to reconcile your changes to your custom `adv3` files with those made in the standard version. The `replace` and `modify` mechanism can help you deal with this problem.

These keywords allow you to make changes to objects and classes that have been previously defined. In other words, you can use the standard `adv3` library, and then make changes to the objects that the compiler has *already* finished compiling. Using these keywords, you can make four types of changes to previously-defined objects: you can replace a function entirely, you can replace an object entirely, or you can add to or change the methods already defined in an object, or you can modify a function.

To replace a function that's already been defined, you simply preface your replacement definition with the keyword `replace`. Following the keyword `replace` is an otherwise normal function definition. The following example replaces the `addToScore()` function defined in `score.t` (part of the standard adv3 library):

```
replace addToScore(points, desc)
{
   if(gPlayerChar.isWorthy)
      libScore.addToScore_(points, desc);
}
```

You can do exactly the same thing with objects or classes. For example, you can entirely replace the `coarseMesh` object defined in sense.t:

```
replace coarseMesh: Material
   seeThru = transparent
   hearThru = transparent
   smellThru = distant
   touchThru = transparent
;
```

Replacing an object or class entirely deletes the previous definition, including all inheritance information and vocabulary. The only properties of a replaced object are those defined in the replacement; the original definition is entirely discarded.

You can also modify an object or class, retaining its original definition (including inheritance information, vocabulary, and properties). This allows you to add new properties and vocabulary. You can also override properties, simply by redefining them in the new definition.

For example, you might want to change one of the standard library responses and add one of your own:

```
modify playerActionMessages
   cannotTurnMsg = '{The dobj/he} just will not turn. '
   shouldNotSpitMsg = 'It's rude to spit in public. '
;
```

Note that no superclass information can be specified in a `modify` statement; this is because the superclass list for the modified object is the same as for the original object.

In a method that you redefine with `modify`, you can use `inherited` to refer to the *replaced* method in the original definition of the object. In essence, using `modify` renames the original object, and then creates a new object under the original name; the new object is created as a subclass of the original (now unnamed) object. (There is no way to refer to the original object directly; you can only refer to it indirectly through the new replacement object.) Here's an example of using `inherited` with `modify`.

```
     class testClass: object
           sdesc = "testClass"
     ;

     testObj: testClass
        sdesc
           {
              "testObj...";
             inherited;
           }
     ;

     modify testObj
        sdesc
           {
             "modified testObj...";
             inherited;
           }
     ;
```

Evaluating `testObj.sdesc` results in this display:

```
      modified testObj...testObj...testClass
```

You can also replace a property entirely, erasing all traces of the original definition of a property. The original definition is entirely forgotten - using `inherited` will refer to the method inherited by the original object. To do this, use the `replace` keyword with the property itself. In the example above, we could do this instead:

```
modify testObj
  replace sdesc
     {
         "modified testObj...";
          inherited;
     }
;
```

This would result in a different display for `testObj.sdesc`:

```
modified testObj...testClass
```

The `replace` keyword before the property definition tells the compiler to completely delete the previous definitions of the property. This allows you to completely replace the property, and not merely override it, meaning that `inherited` will refer to the property actually inherited from the superclass, and not the original definition of the property.

The `modify` keyword can also be used in a function definition. Modifying a function is just like replacing it (using the `replace` keyword), except that the new definition of the function can invoke the old definition of the function (i.e., the definition that's being replaced). This allows the program to apply incremental changes to a function, such as adding new special cases, without the need to copy the full text of the original function.

To invoke the previous definition of the function, use the `replaced` keyword. This keyword is syntactically like the name of a function, so you can put a parenthesized argument list after it to invoke the past function, and you can simply use the replaced keyword by itself to obtain a pointer to the old function. Here's an example.

```
getName(val)
{
  switch(dataType(val))
  {
  case TypeObject:
    return val.name;

  default:
    return 'unknown';
}

// later, or in a separate source module
modify getName(val)
{
  if (dataType(val) == TypeSString)
    return '\'' + val + '\'';
  else
    return replaced(val);
}
```

Note how the modified function refers back to the original version: we add handling for string values, which the original definition didn't provide, but simply invoke the original version of the function for any other type. The call to `replaced(val)` invokes the previous definition of the function, which we're replacing.

Once a function is redefined using `modify`, it's no longer possible to invoke the old definition of the function directly by name. The only way to reach the old definition is via the `replaced` keyword, and that can only be used within the new definition of the function.

iv) Delegated

It is sometimes desirable to be able to circumvent the normal inheritance relationships between objects, and call a method in an unrelated object as though it were inherited from a base class of the current object. For example, you might want to create an object that sometimes acts as though it were derived from one base class, and sometimes acts as though it were derived from another class, based on some dynamic state in the object. Or, you might wish to create a specialized set of inheritance relationships that don't fit into the usual class tree model.

The `delegated` keyword can be useful for these situations. This keyword is similar to the `inherited` keyword, in that it allows you to invoke a method in another object while retaining the same "self" object as the caller. `delegated` differs from `inherited`, though, in that you can delegate a call to *any* object (or class), whether or not the object is related to "self." In addition, you can use an object expression with `delegated`, whereas `inherited` requires a compile-time constant object.

The syntax of `delegated` is similar to that of `inherited`:

```
  return_value = delegated object_expression.property
optional_argument_list
```

For example:

```
book: Thing
  handler = Readable
  doTake(actor) { return delegated handler.doTake(actor); }
;
```

In this example, the `doTake` method delegates its processing to the `doTake` method of the object given by the "handler" property of the "self" object, which in this case is the `Readable` object. When `Readable.doTake` executes, its "self" object will be the same as it was in `book.doTake`, because `delegated` preserves the "self" object in the delegatee.

In the delegatee, the `targetobj` pseudo-variable contains the object that was the target of the `delegated` expression.

d. *Afterword*

There is more to the TADS 3 language than has been described here, but hopefully we have now covered the basics, and once you have mastered those you will be able to glean the rest from the *System Manual*. There's no need to do that until you've worked your way through this guide, although of course if you're burning with curiosity to find out what else is there, there's nothing to stop you!

Chapter Two - A Sample Game

In the next chapter we'll start developing a game that will occupy us for the remainder of this *Guide* (apart from the odd explanatory digression or two). But before we embark on *The Further Adventures of Heidi* we'll start with a very simple two-room game (the goldskull example familiar to TADS 2 users) that provides an overview of how a TADS 3 program fits together. When you are reading later sections, which go into more detail, it may be helpful to have an idea of where the details fit into the general structure of a game. This chapter should help provide that overview (but readers with some experience of other TADS-like languages who find the going a bit too slow might like to skip this chapter and go straight to the next).

The basic requirements for starting out are the TADS 3 Author's Kit and a text editor. If you are using TADS 3 Workbench for version 3.0.13 or later you can use its built-in editor (probably the best option both because it has been specially apdapted for working with TADS 3 code and because its integration with Workbench makes it especially convenient to use); otherwise if all else fails you can use Notepad (or I suppose a really determined UNIX user could use vi), but you may like to consider downloading one of the many free programming editors available on the internet. Information about programming editors that can be used for writing Interactive Fiction may be found at http://www.firthworks.com/roger/editors/index.html.

1. *A Very Simple Game*

We'll start with about the simplest game possible: two rooms, and no objects. (We could conceivably start with only one room, to make things even simpler, but then there would be nothing to do while playing the game; with two rooms, we at least can move between them.)

The basis for the game we shall be developing is the so-called 'advanced' starter game starta3.t, which should be located in the samples subdirectory of your TADS 3 directory. If you are using the TADS 3 Workbench, select New Project, choose the 'advanced' rather than the 'beginner' option, call the new file you are about to create 'goldskull.t' and locate it in whichever directory you want to work (it's probably a good idea to create a new directory called Goldskull or the like for the purpose). Otherwise, if you are not using Workbench, copy starta3.t to your new Goldskull directory and rename it goldskull.t. Again, if you are *not* using Workbench you will need to use your text editor to create a file called goldskull.t3m (in the same location) containing the following:

```
-DLANGUAGE=en_us
-DMESSAGESTYLE=neu
-Fy obj -Fo obj
-o goldskull.t3
-lib system
-lib adv3/adv3
-source goldskull
```

Now open goldskull.t in Workbench (if you’re using Workbench) or else in text editor of your choice; the TADS Compiler will accept an ASCII file produced with any editor. Then remove (or modify as shown below) the definition of startroom, i.e. the lines that read:

```
startRoom: Room 'Start Room'
    "This is the starting room. "
;
```

If you started from starta3.t your file should already contain the vital lines:

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

If not, you will need to add them. You will also need to ensure that your source file contains:

```
gameMain: GameMainDef
    /* the initial player character is 'me' */
    initialPlayerChar = me
;

/* You could customize this if you wished */
versionInfo: GameID
 /* The IFID can be any random set of hexadecimal digits in this format */
   IFID = '5b252939-8c87-0a51-dd3f-eafb1c07da05'
    name = 'Gold Skull'
    byline = 'by A TADS 3 Tyro'
    htmlByline = 'by <a href="mailto:$EMAIL$">
                  $AUTHOR$</a>'
    version = '1'
    authorEmail = '$AUTHOR$ <$EMAIL$>'
    desc = '$DESC$'
    htmlDesc = '$HTMLDESC$'
;
```

Then you can start adding the new code (or adapting the definition of `startroom` that starta3.t already provides):

```
startroom: Room                  /* we could call this anything we liked */
    roomName = 'Outside cave'    /* the "name" of the room */
    desc = "You're standing in the bright sunlight just
    outside of a large, dark, foreboding cave, which
    lies to the north. "
    north = cave         /* the room called "cave" lies to the north */
  ;

+ me: Actor /* This may already be there if you started from starta3.t */
;

cave: Room
    roomName = 'Cave'
    desc = "You're inside a dark and musty cave. Sunlight
    pours in from a passage to the south. "
    south = startroom
;
```

To run this example, all you have to do is compile it with `t3make`, the TADS 3 Compiler, and run it with `t3run`, the TADS 3 Run-time system. If you are using Workbench, this is all handled for you; you can simply choose the 'Compile and Run' from the 'Build' menu (or click the appropriate icons on the task bar). If you're not using Workbench, then on most operating systems you can compile your game by typing this:

```
t3make -d -f goldskull
```

and you can run it by typing this:

```
t3run mygame
```

If you have difficulty getting this to work, consult the README file that came with your distribution. It's possible, for example, that you may need to manually create a subdirectory called obj under you main game directory (Workbench handles this automatically).

Now we'll walk through the sample game line by line.

The `#include` command inserts another sourcefile into your program. The file called `adv3.h` is a set of basic definitions that allows your game to work properly with the adv3 library (note that the library files themselves are *not* included; for a full explanation of this see the article on 'Separate Compilation' in the *Technical Manual*, but there's no need to do that right now). The actual adv3 library files are included in your project by virtue of your goldskull.t3m file (which Workbench will have created for you automatically, if you are using Workbench). You should be able to use these definitions, with few changes, for most adventure games. By incorporating the `adv3` library in your game, you don't need to worry about definitions for basic words such as "the," a large set of verbs (such as "take," "north," and so forth), and many object classes (more on these in a bit).

The line including `en_us.h` is similar; it contains some additional standard definitions to interface with the parts of the library that are specific to the English language. The reason for placing these definitions in a separate file is that it is then much easier to customize TADS 3 to work with other languages.

The line that says `startroom: Room` tells the compiler that you're going to define a room named "startroom". Now, a `Room` is nothing special to the TADS 3 *language*, but the `adv3` *library* that you incorporated defines what a `Room` is. A `Room` is one of those object classes we mentioned. The next line defines the `roomName` for this room. A `roomName` is a short description; for a room, it is normally displayed whenever a player enters the room. The `desc` is the long description; it is normally displayed the first time a player enters the room, and can be displayed by the player by typing "look".[7] Finally, the `north` definition says that another room, called `cave`, is reached when the player types "north" while in `startroom`.

A bit of terminology: `startroom` and `cave` are *objects*, belonging to the *class* `Room`; `roomName`, `desc`, `north`, and the like are *properties* of their respective objects. In the context of TADS programming, an object is a named entity which is defined like `startroom`; each object has a class, which defines how the object behaves and what kind of data it contains. Note that our usage is sometimes a little loose, and we will also use "object" the way a player would, to refer to something in the game that a player can manipulate. In fact, each item that the player thinks of as an object is actually represented by a TADS object (sometimes several, in fact); but your TADS program will contain many objects that the player doesn't directly manipulate, such as rooms.

If you're familiar with other programming languages, you may notice that the program above appears to be entirely definitions of objects; you may wonder where

[7] The default behaviour in TADS 3 is in fact for a room's long description to be displayed *every* time the player character enters the room, though the player can change that behaviour.

the program starts running. The answer is that the program doesn't have an obvious beginning in the code we typed.

TADS 3 employs a style of programming different from that you may have encountered before; this new style may take a little getting used to, but you'll find that it is quite powerful for writing adventure games and simplifies the task considerably. Most programming languages are "procedural"; you specify a series of steps that the computer executes in sequence. TADS, on the other hand, is more "declarative"; you describe objects to the system. While TADS programs usually have procedural sections, in which steps are executed in sequence, the overall program doesn't have a beginning or an end; or rather it does, but these are buried deep inside the adv3 library and taken care of for you.[8]

The reason TADS 3 programs aren't procedural is that the player is always in control of the game. When the game first starts, the library calls a bit of procedural code in your program that displays any introductory text you wish the player to see, then the system waits for a command from the player. Based on the command, the system will manipulate the objects you defined according to how you declare these objects should behave. You don't have to worry about what the player types; you just have to specify how your objects behave and how they interact with one another.

2. *Adding Items to the Game*

Now let's add a few items to the game that can be manipulated by the player, so he can do something besides walk back and forth between our two rooms. We'll add a solid gold skull, and a pedestal for it to sit upon.

```
pedestal: Surface, Fixture
  name = 'pedestal'
  noun = 'pedestal'
  location = cave
;

goldSkull: Thing
  name = 'gold skull'
  noun = 'skull' 'head'
  adjective = 'gold'
  location = pedestal
;
```

Here we've defined two objects, `pedestal` and `goldSkull`.

The `pedestal` belongs to two classes, `Surface` and `Fixture`. This means that it has attributes of both classes; when there's a conflict, the `Surface` class takes precedence, because it's first in the list of classes. Objects of the `Surface` class can have other objects placed on top of them; objects of the `Fixture` class can't be carried. The `goldSkull` belongs to the `Thing` class, which is the generic class for portable objects without any special properties.

Since these objects can be manipulated directly by the player, the player needs words to refer to them. This is what the `noun` and `adjective` properties define. All objects that the player can manipulate must have at least one `noun`. Note that the `goldSkull` has two nouns; they are simply listed with a space between them. Objects

[8] Of course it would in principle be perfectly possible *not* to use the adv3 library and then write purely procedural code in TADS 3, but that's a complication beyond the scope of this introductory guide.

can also have adjectives; these serve to distinguish between objects which have the same noun, but are otherwise optional. A good game will recognize all of the words it uses to describe an object, so if you describe the skull as a "gold skull," you should understand it when the player says "take the gold skull."

Although here we have defined noun and adjective as separate properties, as indeed they are, the English-language part of the TADS 3 library allows a short cut: we can instead define an object's vocabulary – its nouns and adjectives – in the single property `vocabWords`, like so:[9]

```
vocabWords = 'gold skull/head'
```

This brings us to a subtlety. Notice that the `desc` property uses *double* quotes around its strings, but the other properties have *single* quotes. The distinction is that a string enclosed in double quotes is displayed immediately every time it is evaluated, while a string enclosed in single quotes is a string value that can be manipulated internally. Double-quoted strings are displayed automatically as a convenience, since most strings in text adventures are displayed without further processing. (Note that the double quote mark is a separate character on the keyboard, and is not simply two single quote marks.) We'll discuss this distinction further at the end of the next chapter.

These two objects have another new property, `location`. This simply defines the object that contains the object being defined. In the case of the `pedestal`, the containing object is the `cave` room; since the `goldSkull` is on the `pedestal`, its location is `pedestal`. Note that the internal workings of the containment model make no distinction between an object being *inside* another object and the object being *on* another object. This means that an object can't (usefully) be both a `Surface` and a `Container`.[10]

3. *Making the Items Do Something*

The game is still rather bland; it has no puzzles. So, let's introduce a small puzzle. Let's assume that the gold skull wasn't merely left lying around; instead, whoever left it there arranged for a trap to go off if it should be lifted off the pedestal. To implement this, we need to add a *method* to the `goldSkull` object. A *method* is a special type of property which contains code (i.e. a sequence of one or more statements) to be executed; it is very much like a function in C or Pascal. The new `goldSkull` with the method looks like this:

```
goldSkull: Thing
  name = 'gold skull'
  vocabWords = 'gold skull/head'
  location = pedestal

  actionDobjTake()
  {
    "As you lift the skull, a volley of poisonous
    arrows is shot from the walls! You try to dodge
```

[9] In the next chapter we’ll see a way to make this short-cut even shorter.

[10] Although the ComplexContainer class allows us to *simulate* an object being both a Surface and a Container; but that’s something we’ll come to in due course.

```
      the arrows, but they take you by surprise!";

      finishGameMsg(ftDeath, [finishOptionUndo]);
    }
;
```

The method `actionDobjTake` (which stands for "action **D**irect **obj**ect **Take**") is invoked when the player (or any other actor) tries to take the skull. Here, we've simply defined it first to display a message (since the message is enclosed in double quotes, it is displayed immediately upon being evaluated), and then to call a special function called `finishGameMsg` (the argument `ftDeath` shows that we want `finishGameMsg` to end the game with a YOU HAVE DIED message; `finishOptionUndo` offers the player an UNDO option after the death message).

You should note that we didn't just pick the name `actionDobjTake` out of thin air. The `actionDobjTake` method in the `goldSkull` object is called by TADS 3 when the player types a "take" command with `goldSkull` as the direct object. Each verb the player types results in the system calling particular methods in the object or objects named in the command. The naming of these methods is described in more detail later in this guide.[11]

You might wonder why we didn't need a `actionDobjTake` method in our original definition of `goldSkull`, or you might have assumed that the system automatically knows what to do if no `actionDobjTake` is defined for an object. In fact, all objects do need a `actionDobjTake` method, and the system doesn't automatically know anything about it. However, since practically every object has the same `actionDobjTake`, with a few exceptions such as `goldSkull`, it would be extremely tedious to type a `actionDobjTake` method for every object in the game. Instead, we use something called "inheritance." By defining the `goldSkull` to be a member of the `Thing` class, you tell TADS 3 that `goldSkull` "inherits" all of the definitions for `Thing`, in addition to any definitions it makes on its own. The `Thing` class, which appears in the `adv3` library file included at the beginning of the program, defines a `actionDobjTake` method, so anything that is defined to be a `Thing` inherits that definition. However, if something is defined in both `Thing` and `goldSkull`, as `actionDobjTake` is in this example, the definition in `goldSkull` takes precedence - it "overrides" the inherited method.

We actually don't have a very good puzzle here, because there's no way to take the gold skull without dying. So, let's put a rock on the cave floor:

```
smallRock: Thing
  name = 'small rock'
  vocabWords = 'small rock'
  location = cave
;
```

Now, let's change the `actionDobjTake` method of the `goldSkull`.

```
actionDobjTake()
{
  if (location != pedestal ||        /* am I off the pedestal? */
  smallRock.location == pedestal )  /* or is the rock there? */
    inherited;                       /* yes - take as usual */
  else                               /* no - the trap goes off! */
  {
```

[11] In TADS 3 code they often *appear* to be named differently, due to the use of the dobjFor() and iobjFor() macros; but we'll come to them in the next chapter.

```
      "As you lift the skull, a volley
      of poisonous arrows is shot from
      the walls! You try to dodge the
      arrows, but they take you by surprise!";

      finishGameMsg(ftDeath, [finishOptionUndo]);
    }
  }
```

This new `actionDobjTake` first checks to see if the object being taken (the special object `self`, which is the object to which the message `actionDobjTake` was originally sent), which in this case is the gold skull, is already off the pedestal; if it is, we don't want anything to happen, so we invoke the *inherited* handling of `actionDobjTake`. We also use the inherited handling if the small rock is on the pedestal. When we invoke the *inherited* handling, the `actionDobjTake` method that we inherit from our parent class (in this case, `Thing`) is invoked. This allows us to override a method only under certain special circumstances, and otherwise do business as usual. If we don't satisfy one of these two requirements, the volley of poisonous arrows is released as before.

So, the solution to the puzzle is to put the rock on the pedestal before taking the skull, thereby fooling the pedestal into thinking the skull is still there.

As this stands, the player can avoid losing the game, but can't actually win it. To finish the game with a winning ending instead of a deadly one, you can call `finishGameMsg` with `ftVictory` instead of `ftDeath`. If you want to try experimenting for yourself before going on to the next chapter, see if you can make the game end in victory either (a) when the player succeeds in picking up the gold skull, or (b) when he succeeds in leaving the cave with it. For the latter, try putting your extra code in the `enteringRoom(traveler)` method of `startroom`, and testing for the condition `goldSkull.isIn(gPlayerChar)`. If you want to be even more adventurous, try adding another room, say a path through the jungle leading away from `startroom`, and make the player win the game when the player character enters your new room with the skull. However, if you don't feel confident enough to try any of this on your own just yet, no matter, just read on.

This should give you some idea of how a TADS 3 program looks. In the next chapter, we'll start to develop a somewhat larger game, starting once again from the very basics and developing our understanding of how they're normally handled in TADS 3, before going on to add items and puzzles of increasing complexity.

Chapter Three - Starting Out Again - Defining Rooms and Objects

1. *Starting a New Game*

In the previous chapter we saw how to create a very simple TADS 3 game. In this chapter we shall start creating a somewhat more complex game, which will occupy us for the remainder of this guide. Although in the initial stages there will be some overlap with what has gone before, it is important to ensure that the foundations of understanding are securely laid, and in any case we shall shortly be introducing new ways of accomplishing seemingly familiar tasks.

The basis for the game we shall be developing is once again the so-called 'advanced' starter game starta3.t, which should be located in the samples subdirectory of your TADS 3 directory. If you are using the TADS 3 Workbench, select New Project, choose the 'advanced' rather than the 'beginner' option, call the new file you are about to create 'heidi.t' and locate it in whichever directory you want to work (it's probably a good idea to create a new directory called Heidi for the purpose). Otherwise, if you are not using Workbench, copy starta3.t to your new Heidi directory and rename it heidi.t.

Now open the file in your text editor of choice (either through Workbench or through the editor) and remove the definition of startroom, i.e. the lines that read:

```
startRoom: Room 'Start Room'
    "This is the starting room. "
;
```

Next, change the line `location = startRoom` (after the comment `/* the initial location */`) so that it reads `location = outsideCottage`. You might also like to fill in the other fields with something a bit more meaningful, so that the edited file looks something like:

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

versionInfo: GameID
    IFID = '573a8b18-1008-ca66-9580-9a156f82eefa'
    name = 'The Further Adventures of Heidi'
    byline = 'by An Author'
    htmlByline = 'by <a href="mailto:whatever@nospam.org">
                  ERIC EVE</a>'
    version = '1.0'
    authorEmail = 'ERIC EVE <whatever@nospam.org>'
    desc = 'This is an unexciting tutorial game based loosely on
       The Adventures of Heidi by Roger Firth and Sonja Kesserich.'
    htmlDesc = 'This is an unexciting tutorial game based loosely on
       <i>The Adventures of Heidi</i> by Roger Firth and Sonja Kesserich.'

    showCredit()
    {
        /* show our credits */
        "The TADS 3 language and library were created by Michael J.
             Roberts.<.p>
        The original <i>Adventures of Heidi</i> was a simple tutorial game
        for the Inform language written by Roger Firth and Sonja Kesserich.";

         "\b";
    }
    showAbout()
    {
        "<i>The Further Adventures of Heidi</i><.p>
        A Tutorial Game for TADS 3";
    }
;

me: Actor
    /* the initial location */
    location = outsideCottage
;

gameMain: GameMainDef
     initialPlayerChar = me
     showIntro()
     {
       "Welcome to the Further Adventures of Heidi!\b";
     }
     showGoodbye()
     {
       "<.p>Thanks for playing!\b";
     }
     maxScore = 7
     beforeRunsBeforeCheck = nil
;
```

2. *Defining our first Room*

So far, the only really significant thing we have done to the source file is to indicate that the room in which the game will start will be called `outsideCottage`. Our next job is to define this room. As a first attempt, add the following to the end of your source file:

```
outsideCottage: OutdoorRoom
   roomName = 'In front of a cottage'
   desc = "You stand just outside a cottage; the forest stretches east. "
;
```

Be careful to copy this code exactly, including the punctuation, not least the semicolon at the end (by itself on the last line). Also, be careful to note that 'In front of cottage' is enclosed in single quotation marks, and "You stand outside a cottage; the forest stretches east. " in double quotation marks. This distinction is important and must be followed (the significance of the distinction will be explained in more detail on p. 49 below).

If you compile and run the game it should now run, although the game is pretty basic. Since there's only one room in the game you can't actually move anywhere, and there are no objects to manipulate or even examine. About the most interesting thing you can do with the game right now is to **quit** straight away!

Before making things more interesting, let's take a look at the definition of the one room we have defined so far. The first line `outsideCottage: OutdoorRoom` consists of the object name followed (after the colon) to the superclass to which it belongs.[12] The object name is simply the name by which this object will be referred in

[12] An object’s ‘class’ defines how the object behaves; the library defines a number of classes for typical

our code; we could have called it `room101` or `auntieMyrtle`, but it is obviously better to choose something that makes reasonable sense. Note that we have followed the TADS 3 convention of starting an object name with a lowercase letter, while using a capital letter at the start of any subsequent words in the name.

`OutdoorRoom` is the name of the class to which we want this game object to belong. By default an `OutdoorRoom` has ground and sky, but no walls, which is what we want here. Try running the game again and type **examine ground**, **x sky** and **x wall**. Now change `OutdoorRoom` to `Room`, and compile and run the game again and type the same commands. Finally change `Room` back to `OutdoorRoom`. Again note the naming convention, since `OutdoorRoom` names not an object but a *class* it starts with a capital letter.

The next two lines define the *properties* of our OutdoorRoom object:

```
roomName = 'In front of a cottage'
desc = "You stand just outside a cottage; the forest stretches east. "
```

The `roomName` property (a string enclosed in single quotation marks) is the brief title that names the current room in the status line and at the start of a room description. The `desc` property (a double quoted string) is the longer description that is displayed the first time a room is seen, or in response to a **look** command (or every time a room is entered if the game is in **verbose** mode).[13]

Finally, on the last line, is a semicolon by itself; this simply ends the definition of this object. An alternative is to enclose the property list in curly braces thus:

```
outsideCottage: OutdoorRoom
{
   roomName = 'In front of a cottage'
   desc = "You stand just outside a cottage; the forest stretches east. "
}
```

Either form is possible and which you use is largely up to you. There are situations (as we shall see later) in which you have to use braces; in other situations (as we shall again see) the use of the semicolon can make for more compact code.

Note that the semicolon is also used to terminate TADS 3 program statements. This can be a source of confusion because the property definitions look rather like assignment statements, so it can be very easy to slip into writing:

```
outsideCottage: OutdoorRoom
   roomName = 'In front of a cottage';
   desc = "You stand just outside a cottage; the forest stretches east. ";
;
```

If you try to compile this, you'll get an error, because the compiler will now think the definition of `outsideCottage` ends with the `name` property and won't know what to do with the `desc` property. Since this is such a common source of potential confusion it's worth remembering the following golden rule straight away:

A property definition is not a programming statement. Do not end it with a semicolon.[14]

---

game situations, such as OutdoorRoom, but you can also define new classes of your own.

[13] For the difference between single and double-quoted strings see below, p. 51.

[14] This is not strictly one hundred per cent accurate, since if you use the second form of object definition, enclosing the entire list of properties and methods in braces, then it's okay (though still

We have laboured the definition of `outsideCottage` at some length, since the principles involved are common to a great deal of TADS 3 programming, most of which consists in defining objects.

If you read the previous chapter, then all this should be reasonably familiar from the "goldskull" example. Now we come to our first major innovation: although the definition of `outsideCottage` seems simple enough, it can in fact be made a good deal simpler through a feature of the language called 'templates'. A template simply defines a shorthand way of defining the most common properties an object is likely to have. Since every room will have a name and a description the TADS 3 library defines a template that looks like this:

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

This means that if we follow the class name of a Room-type object with a string in single quotation-marks, it will be taken as the `roomName` property of the Room; we can then optionally supply a second single-quoted string as the `destName` property and (if `destName` is supplied) a third single-quoted string as the `name` property (a pair of complications we shan't go into here) and, also optionally, a double-quoted string as the `desc` property.[15] This would allow our room to be defined simply as:

```
outsideCottage : OutdoorRoom 'In front of a cottage'
   "You stand just outside a cottage; the forest stretches east. "
;
```

In other words, when defining a room we can simply follow the class (or class list) with the room name in single quotation marks, followed by the full room description in double quotation (ignoring the `destName` property for now). Since this is generally a far more convenient way of defining objects, it is the way we shall generally adopt from now on. The compiler will, however, complain if you attempt something that does not conform to the template; for example, you would get a compile-time error if you wrote:

```
outsideCottage : OutdoorRoom
   "You stand just outside a cottage; the forest stretches east. "
   'In front of a cottage'
;
```

In other words, the properties must be supplied in the order defined in the template, and must conform to the number and format of properties the template expects. Note, however, that there is nothing magical about laying the code out for this object definition on three lines. So far as the compiler is concerned, it could have been written:

```
outsideCottage : OutdoorRoom 'In front of a cottage' "You stand just outside
a cottage; the forest stretches east. ";
```

unnecessary) to end property definitions with a semicolon. However, it's simpler and easier to get into the habit of never ending a property definition with a semicolon.

[15] The use of the destName property will be discussed below, on p. 158.

It is just that the three-line version is more readable to the human eye, and makes for more legible code. Thanks to templates, though, there are cases in which it is both feasible and legible to code an entire object on a single line.

3. *Adding an Object to the Room*

It is time we added an object to our sample game. If you run the game again and try typing **examine cottage** you'll be told that:

The word "cottage" is not necessary in this story.

But since our minimalist room description mentions the cottage, our game ought to be able to do a bit better than that. This suggests that the first thing we need to add to our game is a cottage. If we defined it in full, our first attempt might look like this:

```
cottage : Thing
   vocabWords = 'pretty little cottage/house/building'
   name = 'pretty little cottage'
   desc = "It's just the sort of pretty little cottage that townspeople
    dream of living in, with roses round the door and a neat little
    window frame freshly painted in green. "
   location = outsideCottage
;
```

As we shall see in a moment, this can be simplified using the appropriate template and the + syntax, but writing it out in full has the merit of explicitly re-introducing two important properties, `location` and `vocabWords`. The first of these, as its name suggests, defines the location of an object (in this case, which room it's in); more generally it defines what the immediate parent of an object is in the object tree. You should normally avoid manipulating the location property in programming statements (but here we have a property definition, not a statement). The second property, `vocabWords`, lists the words the player can use to refer to the object. In this definition, the final group of words separated by slashes (`cottage/house/building`) are the *nouns* by which this object may be known, whereas the first two, separated by spaces, are the *adjectives*. This means that if you now compile the game and run it, with the cottage added, you'll find that you can **examine building**, **examine little house**, **examine pretty little cottage**, but not **examine house cottage**, or **x building house**.

In practice you would hardly ever define all those properties explicitly, instead you'd make use of the standard Thing template (which can be used not only for Thing objects but for objects whose classes descend from Thing, which is the great majority of physical objects in the game). Using this template, the definition becomes:

```
cottage : Thing 'pretty little cottage/house/building'
   'pretty little cottage'  @outsideCottage
   "It's just the sort of pretty little cottage that townspeople dream of
living in, with roses round the door and a neat little window frame freshly
painted in green. "
;
```

Note the form of this definition, since it is *very* common in TADS 3. After the superclass (or superclass list) comes first the list of vocabulary words in single quotes, in the form 'adjective1 adjective2 noun1/noun2/noun3', then the name in single quotes, then the location immediately preceded by an @ sign, and then the description in double quotes (left till the end since it is likely to be the longest element). The list of vocabulary words should always include at least one noun, but may otherwise contain as many adjectives and alternative nouns as you care to define. Ideally, what you need to aim for is a list of words that will include most of those that a player is likely to type to identify the object, while at the same time being sufficiently distinct from the words used to identify other objects that the parser will not have too hard a time trying to figure out which object is meant.

The @location element is optional in this template, so you could simply define, for example:

```
cottage : Thing 'pretty little cottage/house/building'
   'pretty little cottage'
      "It's just the sort of pretty little cottage that townspeople dream of
living in, with roses round the door and a neat little window frame freshly
painted in green. "
;
```

The only problem with this is that there's now nothing to say where the cottage is located; the game will still compile but the cottage will have disappeared. One way to bring it back would be to use the + notation, so that the cottage could be defined:

```
+ cottage : Thing 'pretty little cottage/house/building'
   'pretty little cottage'
   "It's just the sort of pretty little cottage that townspeople dream of
living in, with roses round the door and a neat little window frame freshly
painted in green. "
;
```

The + is just a shorthand way of saying "set the location property of this object to the nearest previous object in the current source file not preceded by a +". The + shortcut can be used to nest to any level, so that if we began an object definition with `++ myObj`, the location property of `myObj` would be set to the nearest preceding object beginning with a single + and so on. This allows for very compact code for defining nested objects, e.g.:

```
study : Room 'study' "A large desk stands under the window. ";
+ desk : Heavy, Surface 'desk' 'desk' "This large desk has a single drawer.
";
++ drawer : Component, OpenableContainer 'drawer', 'drawer' "It looks like
it should open easily. ";
+++ redPencil : Thing 'red pencil' 'red pencil' "It's a bit blunt. ";
+++ bluePencil: Thing 'blue pencil' 'blue pencil' "It's been sharpened
recently. ";
```

In this case both the red pencil and the blue pencil will be inside the drawer, the drawer inside the desk, and the desk inside the study.

But, to return to our cottage, if you change its definition to the latest version above and recompile and run the game, you should find it still works the same, but the way it works isn't quite what we want. For one thing, the room description already mentions the cottage (that's why we created a cottage object in the first place), so it's rather superfluous for the game to add "You see a pretty little cottage here." More seriously, if you type **take cottage** and then **inventory** (or **i**) you'll find that you're carrying a pretty little cottage, which should probably count as murder of mimesis in the first degree.[16]

The problem here is that `Thing` is the most generic class of object in the library. For objects that you want the player character to be able to pick up and carry around it's often fine, but for things that are fixed in place or otherwise not intended to move, it's not the best class to use. We could use the Fixture class to fix the present example. Try changing `Thing` to `Fixture` in the definition of cottage and recompiling the game. Then run it again. You'll see that the game no longer displays "You see a pretty little cottage here" and that you can no longer pick the cottage up. This is just about what we want (at least for now), but there's a couple of further refinements we could add.

Firstly, the main reason for adding the cottage was that a cottage was mentioned in the room description, so the player ought to be able to refer to it. So far, we have no other use for the cottage object. In effect, the cottage is purely decorative, part of the scenery but not otherwise part of the game. For this purpose the library defines a `Decoration` class, and that might be the one to use here.

Secondly, since the cottage is purely decorative (at least at this stage) we probably won't need to refer to it anywhere else. We can therefore make it an *anonymous* object, i.e. one to which we do not give an object name. Such an object can simply be defined with its superclass name (or list). So we can finally redefine our cottage as follows:

```
+ Decoration 'pretty little cottage/house/building' 'pretty little cottage'
   "It's just the sort of pretty little cottage that townspeople dream of
living in, with roses round the door and a neat little window frame freshly
painted in green. "
;
```

Try this, and you'll see that the game now simply tells you that "The pretty little cottage isn't important" if you try to do anything with it other than examine it. For now, this is just what we want.

You'll note that the description of the cottage includes a door, a window and some roses. It's always possible that a player may try to examine these; so as an exercise you could try adding further Decoration objects to represent them.

In the present chapter we have learned the basics of defining room objects and other objects. Progress may have seemed slow, but these are the basics that apply to all objects in the game, so we should be able to make more rapid progress from now on. In the next chapter we'll make our game a little more interesting by adding some more rooms and objects.

4. *Tying Up Some Loose Strings*

The two objects we have defined so far have included both double-quoted and single-quoted strings in their property definitions. To the seasoned TADS 2 programmer the distinction will need little further explanation. An author coming from Inform will be partly prepared and partly misled by the way in which single and

[16] Mimesis comes from the Greek word μίμησις – meaning ‘imitation’ or ‘representation by means of art’; in a literary or Interactive Fiction context it refers to making one’s creation a reasonable representation of the real world, which carrying a cottage around wouldn’t be.

double-quoted strings work in that language. Other readers may be totally mystified. At least a brief attempt at explanation is due at this point, since the distinction is fairly basic to TADS programming.

As a first approximation, a single-quoted string is simply a string constant, whereas a double-quoted string is a shorthand form of a statement that displays the string. That is to say the statement

```
        "To err is human; to make a total mess-up requires a computer. ";
```

is equivalent to the statement:

```
        say('To err is human; to make a total mess-up requires a computer. ');
```

It follows, as a first approximation, that a single-quoted string can be used wherever it makes sense to use a string constant, while a double-quoted string can be used wherever it's legal to write a statement. Thus a single-quoted string can be passed as an argument to a function, used in an assignment statement, or manipulated with the various string functions, but a double-quoted string cannot.

The main confusion comes about because a definition such as

```
widget : Thing 'widget' 'brass widget'
  desc = "It's a brass widget"
;
```

might erroneously lead you to suppose that you could subsequently change the desc property of the widget by a statement such as

```
desc = "It's a silver widget";
```

But this code would generate a compiler error. The desc property should be regarded, not as holding a string constant, but a routine that prints a string constant, so that the definition is effectively equivalent to:

```
widget : Thing 'widget' 'brass widget'
        desc  { say('It\'s a brass widget'); }
;
```

It is thus almost as if a property holding a double-quoted string were in reality a method that displays a string, despite its syntactic appearance (by the way, note that when we define a *method* in TADS 3 we do not include the = sign).

The difference in the way the two kinds of string are employed in object definitions is that single-quoted strings are generally used for single words or short phrases that will generally be displayed as part of a longer message (such as the name property), whereas a double-quoted string is generally used for properties that are expected to hold possibly quite lengthy text, usually one or more complete sentences, that will always be displayed just as they are (such as the full description of an object or room). One further key difference between single-quoted and double-quoted strings, and maybe the most important selection criterion in the library itself, is that the value of a single-quoted string can be inspected and manipulated, whereas a double-quoted string can really only be displayed. So, for example, if it's going to be necessary to look inside a string to see if it starts with a vowel, then we'll definitely want the single-quoted version.

As of TADS 3.1.0 both single and double-quoted strings may contain embedded expressions enclosed in double angle-brackets (<< >>). Such embedded expressions may evaluate to a number, a double-quoted string or a single-quoted string (or nothing at all, i.e. nil). This means that the statement

```
"The rain in Spain stays <<someExpression>> in the plain.";
```

is equivalent to

```
say('The rain in Spain stays ' + someExpression + ' in the plain.');
```

Where `someExpression` could, for example, be a function call or another method or property on the same or a different object. Not only does this allow a double-quoted string to print variable text, it allows it to call a method that may have all sorts of other useful side-effects such as changing the game state, a trick we shall be using more than once in what follows.

One can have a single-quoted string by itself as a statement, at least the compiler won't complain about it, but it will do absolutely nothing when the program is run.

A final example may help to make this all a bit clearer. Here's the definition for a widget that changes from brass to silver when it is picked up:

```
widget : Thing 'widget' 'brass widget'
  "It's a <<metal>> widget. "
  dobjFor(Take)
  {
    action()
    {
      name = 'silver widget';
      metal = 'silver';
      inherited;
    }
  }
 metal = 'brass'
;
```

With such an object defined, one could obtain the following transcript:

You see a brass widget here.

>**x widget**
It's a brass widget.

>**take widget**
Taken.

>**x it**
It's a silver widget.

>**i**
You are carrying a silver widget.

One final point about strings: in TADS a string that will be used to display a complete message (as opposed to an isolated word or phrase) should always end with a space (or newline) just before the closing quote, to allow for the possibility that something may be displayed directly after it. For a newline, insert `\n` in your string. For a newline followed by a blank line use `\b` or `<.p>`; the latter form ensures that only one blank line will appear (even if several `<.p>` tags occur in succession), whereas the former, `\b`, may result in several blank lines, depending on what is printed next. If you *want* several blank lines, then you need to use `\b`.

And one final point overall. You may have noticed that the above example used something called `dobjFor(Take)` followed by a method called `action()` enclosed within outer braces. If you followed the goldskull example in the previous chapter, you might have expected to see a method called `actionDobjTake()` here. In fact, the two ways of doing it are exactly equivalent. Technically, `dobjFor(Take)` is a *macro*, which the *preprocessor* expands into the code the compiler actually sees. The effect in this case is that what the compiler actually sees here is a method called `actionDobjTake`, exactly as before. Although a macro is usually meant to be a kind of shortcut, while in this case it actually makes the code a little more verbose, the use of the `dobjFor` and `iobjFor` macros in TADS 3 programming is so common that this is the style we shall follow from now on.[17]

[17] At this stage it's not really necessary to go into the workings of the preprocessor. When you're ready to learn about it in more depth, the full details can be found in the *System Manual*.

Chapter Four - Moving Around

1. *Basic Travel*

The next step is to expand the map to a few more locations (rooms) so we can start moving around. We'll begin by adding the other three locations that feature in the original *Adventures of Heidi*. We have already covered most of what we need to know in order to do this. Add the following code to the end of the existing program. An explanation of new features follows.

```
forest : OutdoorRoom 'Deep in the Forest'
   "Through the deep foliage you glimpse a building to the west.
    A track leads to the northeast, and a path leads south. "
    west = outsideCottage
    northeast = clearing
;

clearing : OutdoorRoom 'Clearing'
   "A tall sycamore tree stands in the middle of this clearing.
    One path winds to the southwest, and another to the north. "
    southwest = forest
    up = topOfTree
    north : FakeConnector {"You decide against going that way right
         now. "}
;


+ tree : Fixture 'tall sycamore tree' 'tree'
    "Standing proud in the middle of the clearing, the stout
     tree looks like it should be easy to climb. "
;

topOfTree : OutdoorRoom 'At the top of the tree'
   "You cling precariously to the trunk, next to a firm, narrow
    branch. "
    down = clearing
;
```

The room definitions and the definition of the tree object should need little explanation. The important new concept that has been introduced here is that of a *travel connector*. A travel connector is an object that controls what happens if an actor attempts to travel via it. To define what happens when an actor tries to move in a certain direction we must attach a travel connector to the appropriate direction property. For example, to define what happens when the player character is in the forest and the player types **west** we attach the connector called `outsideCottage` to the `west` property of `forest`. You may object that `outsideCottage` is simply a room, the room we started by defining; but Rooms are in fact a special kind of `TravelConnector`, connectors that point to themselves as destination. Traveling via a *Room* thus means traveling to that *Room*. So if we want movement to take place directly from one room to another, we simply set the appropriate direction property to the destination room. Note that unlike TADS 2, in TADS 3 the direction properties `northwest`, `northeast`, `southwest`, and `southeast` must be spelled out in full; the other direction properties you will commonly use are `north`, `south`, `east`, `west`, `up`, `down`, `in` and `out`.

You have probably noticed that the `north` property from the `clearing` uses a different kind of connector, a `FakeConnector`. A `FakeConnector` is what it sounds like, a connector that only appears to go somewhere. An attempt to travel via a `FakeConnector` results in its `travelDesc` message being displayed without any travel actually taking place. One use of a `FakeConnector` might be to create 'soft boundaries' to your map, to make it look as if it extends further than it really does.[18] But in this case we're using a `FakeConnector` because the room description mentions a path to the north, which we shall eventually want to implement, but do not wish to implement yet.

The code using this connector would have looked more that using rooms as connectors if we had defined the `FakeConnector` as a separate object thus:

```
fakePath : FakeConnector
   travelDesc = "You decide against going that way right
         now. "
;
```

The clearing would then be defined with

```
north = fakePath
```

What we in fact did was to make `fakePath` both an *anonymous object* and a *nested object* (all nested objects are in fact anonymous, though the reverse is not true). A nested object is simply an object whose definition is nested inside another object definition. In this case the definition of the `FakeConnector` is nested within the definition of the `clearing`. The definition of a nested object must be enclosed within braces (and not terminated with a semicolon). `FakeConnector` uses a template for which a double-quoted string is its `travelDesc` property (the message that displays when one tries to travel via that connector). The definition of the `north` property of `clearing` is thus a convenient shorthand way of saying 'travel north from here is via an anonymous object of class `FakeConnector` whose `travelDesc` property is "You decide against going that way right now. " Although this `FakeConnector` has no name of its own, it can be referred to as `clearing.north`, i.e. the value of the `north` property of the `clearing` object. Since this kind of shortcut definition is exceedingly common in TADS 3 it is worth introducing at this early stage. We shall meet several more examples as we go on to develop the game.

If you compile and run the game as it is it will look as if nothing has changed from the previous chapter; the new rooms we have added won't appear. The reason for this (which you've probably guessed already) is that we haven't added a connector out of the original `outsideCottage` room (a bug waiting to happen when adding more rooms to an already complex map). This is easy enough to put right; just add the following to the definition of `outsideCottage`, between the room description and the terminating semicolon:

```
east = forest
```

The game should now work as expected.

[18] Quite closely related to the FakeConnector is the DeadEndConnector, used to simulate aborted travel; for details see the *TADS 3 Tour Guide* and *Library Reference Manual*.

2. *Climbing the Tree – Remapping Behaviour*

Since the player will encounter a tree in the clearing, and since examining the tree will tell the player that the tree looks climbable, the player will probably try to climb the tree. But at the moment, the command **climb the tree** will result in the game responding, "That is not something you can climb." What we need to do is to modify the tree object so that trying to climb it has the same effect as typing **up.** The simplest way to achieve this is to add the following to the definition of `tree`:

```
dobjFor(Climb) remapTo(Up)
```

I.e. replace **climb tree** with an **up** command. Both `dobjFor` and `remapTo` are macros that expand to more complex code, but that need not detain us here. What the construction means is "when the current object (in this case the tree) is the direct object of a **climb** command, replace this action with what would have happened if the player had simply typed **up**" (or, as someone used to Inform 7 might say, "instead of climbing the tree, try going up").

Since we've just introduced some basic but very important concepts here let's pause to take a closer look at some of them. Whenever you see `dobj` in TADS 3 it'll be an abbreviation for **d**irect **obj**ect. A direct object is the object a command principally works on: the direct object of the command **climb tree** is thus the tree. We define `dobjFor(Whatever)` on an object whenever we want to tell the game what to do when that object is the direct object of a Whatever command. Usually that'll be more complicated than the example we've given here; this example is about as simple as it gets (don't worry, we'll be coming to more complex examples soon enough). The first part of this piece of code `dobjFor(Climb)` means "we're about to define what to do in response to a **climb tree** command" (it means this because we're defining it on the tree); the second part, `remapTo(Up)`, goes on to say *what* we want to do in this case, namely execute an **up** command instead.

This is as simple as it gets because the action we're remapping to is the simplest kind of action: a command with *no* objects. If we wanted to remap to another command consisting of a verb followed by a direct object we could just list the verb and the object. For example we could have used TravelVia and the name of the connector via which we want the player character to travel:

```
dobjFor(Climb) remapTo(TravelVia, topOfTree)
```

This illustrates a couple of useful things: first, how to use `remapTo` in the more general case where the verb takes a direct object, and second, how to use `TravelVia` with a travel connector to carry out travel. It also illustrates once again that a room is a kind of travel connector leading to itself. However, having pointed all this out, we'll revert to the first version, which is what we actually want here. So if you changed your code to try out `TravelVia`, before going on change it back so it reads:

```
dobjFor(Climb) remapTo(Up)
```

3. *Making Life More Problematic*

So far the game allows the player to walk from the cottage to the clearing and then climb the tree, but this is not particularly challenging. The time has come to add a puzzle: let's suppose that in order to climb the tree, Heidi first needs to fetch a chair and stand on it. To make this a puzzle we must first prevent Heidi climbing the tree when she's standing on the ground. To do this we must change the definition of `clearing.up`. As a first attempt, we'll use a close relative (in fact the parent) of the `FakeConnector`, namely the `NoTravelMessage`. Modify the clearing object so that its `up` property is now defined as follows:

```
up : NoTravelMessage {"The lowest bough is just too high for
   you to reach. "}
```

Now recompile the game and try both going up from the clearing and climbing the tree. Both attempts should be foiled in exactlty the same way. If we had remapped **climb tree** to `TravelVia, topOfTree` instead of `Up` this would not have worked; the player could have bypassed our puzzle by typing **climb tree** instead of **up**.

That was the easy part. The tricky part is creating a chair object that will enable Heidi to climb the tree. The first thing we need is somewhere to put it; the most likely place you'd find a chair is probably inside the cottage. For the moment we'll define the inside of the cottage as:

```
insideCottage : Room 'Inside Cottage'
  "You are in the front parlour of the little cottage. The door out
    is to the east. "
   out = outsideCottage
   east asExit(out)
;
```

The only new element here is the `asExit` macro. The cottage lies to the west of the starting postion, so to get from inside the cottage back outside the player might type either **out** or **east**. The `asExit()` macro make going **east** the same as going **out**, but without having **east** listed as a separate exit (either in the status line or in response to an **exits** command). This allows us to make two directions lead to the same destination without misleading the player into supposing that they are two separate exits, instead of two synonyms for the same exit.

Note too that since `insideCottage` is an indoor room, we have defined it to be of class `Room` rather than class `OutsideRoom`. To make this room accessible at all we should add the following to the definition of `outsideCottage`:

```
        in = insideCottage
        west asExit(in)
```

Now recompile the game and you should be able to get inside the cottage by typing either **enter** or **in** or **west** (or **w**). But one thing the player might equally well try, namely **enter cottage** won't work.

The obvious way to fix this on the basis of what we've done before is to add the following to the definition of `cottage`:

```
        dobjFor(Enter) remapTo(In)
```

This will certainly do the job, but it's more work than we need, since the TADS 3 library provides an `Enterable` class to handle just this kind of situation. All we need do, in fact, is to change the definition of cottage to:

```
+ Enterable ->insideCottage 'pretty little
   cottage/house/building' 'pretty little cottage'
   "It's just the sort of pretty little cottage that townspeople dream of
living in, with roses round the door and a neat little window frame freshly
painted in green. "
;
```

This introduces a new template element: `->insideCottage`. In this instance the `->` points to the `TravelConnector` that is traversed when the Enterable is entered. Remember that a room is a kind of TravelConnector, and that travelling via a room is the same as travelling to a room, so for now the command enter cottage will take Heidi to the inside of the cottage (we'll be changing that later when we give the cottage a locked front door). An alternative to using the `->connector` syntax would have been to define the connector property explicitly with:

```
        connector = insideCottage
```

Whether you prefer this as being more readable is up to you.

Now that we have somewhere to put the chair, we can start defining it. What we need is something that we can carry around and stand on (but not both at the same time!). So it must be something that can contain an actor while appearing as an object inside a room. In TADS 3 this kind of object is called a `NestedRoom`. The TADS 3 library includes a subclass of `NestedRoom` called `Chair` that does just the job (a `Chair` is something you can sit on or stand on but not lie on):

```
+ chair : Chair 'wooden chair' 'wooden chair'
  "It's a plain wooden chair. "
;
```

There's one way we can improve the behaviour of this chair before we even think about using it to climb the tree. When Heidi is sent into the cottage, the game displays the plain vanilla default message "You see a wooden chair here." We can improve on this by adding the following property definition to the chair object:

```
initSpecialDesc = "A plain wooden chair sits in the corner. "
```

The `initSpecialDesc` property defines how the object will be described in a room description before the object has been moved (if we wanted to, we could override the conditions under which `initSpecialDesc` was displayed, but that's a complication we won't tangle with for now).

Now try compiling and rerunning the game. You should find that the chair now behaves just as one would expect: you can sit or stand on it (but not lie on it), you can also take it, but you can't take it while you're sitting or standing on it, and you can't sit or stand on it while you're carrying it.

But, as you will discover, the chair still doesn't help Heidi climb the tree. The problem is that we defined the connector on `clearing.up` as a `NoTravelMessage`, which blocks travel under all circumstances. What we need is a connector that allows Heidi to pass only when the chair is at the foot of the tree, i.e. in the clearing. One type of connector appropriate to this task is a `OneWayRoomConnector`, since this possesses methods to control the conditions under which travel is permitted. We could define it thus:

```
up : OneWayRoomConnector
  {
    destination = topOfTree
    canTravelerPass(traveler) { return chair.isIn(clearing); }
    explainTravelBarrier(traveler)
     { "The lowest bough is just too high for
          you to reach. "; }
  }
```

The `canTravelerPass()` method allows travel if and only if it returns true, which in this case will happen if and only if the chair is in the clearing. If travel is disallowed, the method `explainTravelBarrier()` is called to explain why not. In this case we just display a suitable message.

Before we carry on with refining this, let's digress to another matter. The connector we've just defined is defined on the `up` property of `clearing`. This might lead us to suppose that we could have defined a slightly more general version of it by defining:

```
up : OneWayRoomConnector
  {
    destination = topOfTree
    canTravelerPass(traveler) { return chair.isIn(self); }
    explainTravelBarrier(traveler)
     { "The lowest bough is just too high for
          you to reach. "; }
  }
```

Here we have simply changed `chair.isIn(clearing)` to `chair.isIn(self)`, on the assumption that it will effectively mean the same thing. But it won't, since in the context in which we've defined it, `self` refers not to the clearing, but to the nested `OneWayRoomConnector` we've just defined on one of its properties. This is a fatally easy easy mistake to make, and raises the question whether there is a *right* way for a nested object like the anonymous `OneWayRoomConnector` in this example to refer to its host object. There is: what we actually need is `lexicalParent`. We could correctly write the previous example as:

```
up : OneWayRoomConnector
  {
    destination = topOfTree
    canTravelerPass(traveler) { return chair.isIn(lexicalParent); }
    explainTravelBarrier(traveler)
     { "The lowest bough is just too high for
          you to reach. "; }
  }
```

This is now equivalent to writing `chair.isIn(clearing)`, but using `lexicalParent` makes it immediately obvious what the intention is (as opposed to having to check that `chair` refers to the enclosing object).

If you now recompile the game and try it again, you'll find that it now works after a fashion, but that it's less than ideal. There are still several things we should tidy up.

One thing we might like to do is to display a suitable message when the player character climbs off the chair and up the tree, rather than just have Heidi suddenly transported from the chair to the top. There is a `TravelMessage` class that allows a message to be displayed while traveling, but we have already defined the connector to be a `OneWayRoomConnector`. Since, however, the `TravelMessage` class inherits all the methods we have already used, we can simply change `OneWayRoomConnector` to `TravelMessage` and add the following property:

```
travelDesc =  "By standing on the chair you just manage to reach the lowest
  bough and haul yourself up the tree.<.p>"
```

The `<.p>` just adds a paragraph break (blank line) after the message. We could do the same with `\b`, except that using `<.p>` ensures we don't get multiple blank lines should the next message start with `<.p>`. The connector should now look like this:

```
up : TravelMessage
  {
    destination = topOfTree
    canTravelerPass(traveler) { return chair.isIn(lexicalParent); }
     explainTravelBarrier(traveler)
     { "The lowest bough is just too high for you to reach. "; }
     travelDesc =  "By standing on the chair you just manage to
     reach the lowest bough and haul yourself up the tree.<.p>"
  }
```

Recompile the game and try it again. You will soon encounter another small problem: the game now describes Heidi as using the chair to reach the bough whether she's standing on the chair or still on the ground when the **climb tree** or **up** command is issued. You might think this is okay on the grounds that if the player has made Heidi carry the chair to the clearing he's probably figured why, so we don't need to make Heidi *explicitly* stand on the chair before climbing the tree. But there's another problem: the chair is counted as being *in* the clearing even if Heidi is still carrying it, so this code would allow Heidi to use the chair to climb the tree while she's still holding the chair. It would be better, then, to check that Heidi is actually on the chair (which she can't be if she's carrying it) before allowing her to climb. We can achieve this by changing the `canTravelerPass` method to:

```
canTravelerPass(traveler) { return traveler.isIn(chair); }
```

We don't also need to test that the chair is in the clearing, since it already *must* be if Heidi is on the chair when this connector is available to her.

Now everything should work reasonably well, except that the game will now allow Heidi to climb the tree from the chair even if she's only sitting on the chair, and not standing on it. Again, we may not think this matters very much in practice, but if we do, there are various ways we could go about fixing it. Perhaps the simplest for now is to add the condition that Heidi must be standing to the `canTravelerPass()` method, which finally gives us:

```
clearing : OutdoorRoom 'Clearing'
          "A tall sycamore tree stands in the middle of this clearing.
           One path winds to the southwest, and another to the north."
           southwest = forest
           up : TravelMessage
          {  ->topOfTree
             "By clinging on to the bough you manage to haul yourself
             up the tree. "
             canTravelerPass(traveler)
                { return traveler.isIn(chair) && traveler.posture==standing; }
             explainTravelBarrier(traveler)
                { "The lowest bough is just out of reach. "; }
          }
          north : FakeConnector {"You decide against going that way right
            now. "}
;
```

If there were several objects that could be used for Heidi to stand on, the `canTravelerPass(traveler)` method would only become a little more complicated, e.g.:

```
       canTravelerPass(traveler) {
         return traveler.location is in (chair, crate, stepladder) &&
           && traveler.posture == standing;
       }
```

Since an just out-of-reach bough is mentioned when the player tries to get Heidi up the tree without the aid of the chair, we might want to add that bough somewhere. The slight complication is that the bough will be out of reach if Heidi is standing on the ground, but not if she's standing on the chair. The `OutOfReach` class handles this type of situation; you could place the following code immediately after the definition of the tree object:

```
++ bough : OutOfReach, Fixture 'lowest bough' 'lowest bough'
    "The lowest bough of the tree is just a bit too high up for you
     to reach from the ground. "

   canObjReachContents(obj)
   {
     if(obj.posture == standing && obj.location == chair)
        return true;
     return inherited(obj);
   }
   cannotReachFromOutsideMsg(dest)
   {
    return 'The bough is just too far from the ground for you to reach. ';
   }
;
```

Admittedly this doesn't allow Heidi to interact very interestingly with the bough even if she is standing on the chair; she can touch the bough which she can't do from the ground, but that's about it. It might be more interesting if on the bough was concealed an object that Heidi needed to find, but this is a step further than we need to go for this game (but you're welcome to experiment with it if you wish!).

One final point: using one object (like the chair here) to gain access to a connector (like the way up the chair) is a fairly common situation in Interactive Fiction. Often, however, it turns out to be a bit more complicated to implement than the example we have worked throught here. You don't need to worry about that just yet – there's plenty more to do in this guide first – but if when you try to implement something similar in your own game you find TADS 3 doing its best to frustrate you at every turn, you'll also find that help is at hand in the article on 'Using NestedRooms as Staging Locations' in the *Technical Manual*.

4. *Rewarding the Effort*

If the player has gone to all this trouble to reach the top of the tree, perhaps he or she deserves some sort of reward for it. One way we can do this is to add some points to the player's score. This can be done with the statement:

```
addToScore(points, 'reason for awarding points');
```

In this case we might have:

```
addToScore(1, 'reaching the top of the tree.');
```

The two problems we need to solve now are (a) where best to put this statement and (b) how to prevent the player from accumulating a huge (if boring) score by repeatedly going up the tree – the point should be awarded first time round only.

It would possible to code this on the `TravelMessage` object Heidi has to travel via when she climbs the tree, but since what we want to do is to check for Heidi arriving at a the top of the tree regardless of how she gets there, the best solution is to make use of the `enteringRoom` method of the `topOfTree` room. The library code already keeps track of which rooms have been visited by setting their `seen` property, so we can use this to ensure that the point is awarded only the first time Heidi reaches the top of the tree. The revised `topOfTree` room then looks like this:

```
topOfTree : OutdoorRoom 'At the top of the tree'
   "You cling precariously to the trunk, next to a firm, narrow branch."
    down = clearing
    enteringRoom(traveler)
    {
      if(!traveler.hasSeen(self))
          addToScore(1, 'reaching the top of the tree. ');
    }
;
```

Being awarded a point is all very well, but it may all seem pretty pointless if that's all that happens when Heidi arrives at the top of the tree. At the very least she should find something interesting there. Since the room description for the top of the tree mentions a branch, that may be the first thing to add. Then perhaps we could place a bird's nest on the branch (in the original *Adventures of Heidi* the object was to replace the bird's nest, complete with fledgling, to the branch), and finally we could place a worthwhile find in the nest.

Before turning over the page to see how this guide does it, you could have a go at implementing these extra objects yourself. Remember that the branch will need to be a `Supporter` so you can put things on it, and the nest a `Container` so you can put things in it. Remember too that you'll need to make sure that Heidi can't pick up the branch – after all it's part of the tree and fixed in place (if in doubt, look at the pedestal in the 'goldskull' game). Then put something interesting in the nest, and see if you can get your revised game to compile and run.

Here's how this guide does it (this code should be placed immediately after the definition of `topOfTree`).

```
+ branch : Surface, Fixture 'branch' 'branch'
  "The branch looks too narrow to walk, crawl or climb along, but firm
   enough to support a reasonable weight. "
;

++ nest : Container 'bird\'s nest' 'bird\'s nest'
  "It's carefully woven from twigs and moss. "
;

+++ ring : Thing 'platinum diamond ring' 'diamond ring'
  "The ring comprises a sparkling solitaire diamond set in platinum. It
    looks like it could be intended as an engagement ring. "
;
```

Note the use of the + notation to nest (no pun intended) each item in the preceding one. We make the branch a `Surface` so that we can put things *on* it and a `Fixture` so that it's fixed in place; this illustrates how the same object may inherit from more than one superclass (but note that the same object can't be both a `Surface` and a `Container`). The nest is made a `Container` so we can put something *in* it. Internally there's not a lot of difference; the `location` property of `ring` is set to `nest`, and the `location` property of `nest` to `branch`. The difference lies in the way the library code describes the situation (in or on) and the type of commands it will respond to (**put in** or **put on**), as you'll find if you add the code and play with the new version of the game.

You'll probably also find that the discovery of the ring seems rather bland and bathetic,[19] since as soon as Heidi arrives at the top of the tree the game announces "On the branch is a bird's nest (which contains a diamond ring)." It would be more interesting if she had to work a little to find that ring. Besides, one might suppose that the ring would at first be hidden among the twigs and moss that make up the nest. The first step towards making things more interesting, then, is to remove the +++ from in front of the definition of the ring (so that it starts life outside the game world altogether) and then code the nest to respond to a **search** or **look in** command. This code must first check that we haven't already found the ring, and then, if we haven't, it should move the ring into the nest and report the find. While we're at it we might as well award the player a point for the discovery. The appropriate code looks like this:

```
++ nest : Container 'bird\'s nest' 'bird\'s nest'
  "It's carefully woven from twigs and moss. "
  dobjFor(LookIn)
  {
    action()
    {
       if(ring.moved)
       {
         "You find nothing else of interest in the nest. ";
         exit;
       }
       ring.moveInto(self);
       "A closer examination of the nest reveals a diamond ring inside! ";
       addToScore(1, 'finding the ring');
    }
  }
  dobjFor(Search) asDobjFor(LookIn)
;
```

[19] I.e. “anticlimatic”; “bathetic” is *not* a mis-spelling of “pathetic”!

You'll remember that dobj is short for direct object, so that when we define `dobjFor(LookIn)` on an object we're defining what should happen when that object is the direct object of a **look in** command. In this case, though, what comes after `dobjFor` looks rather more complicated than our previous example.

Let's take the simpler part first. Towards the end of the nest object we have written `dobjFor(Search) asDobjFor(LookIn)`. That `asDobjFor()` is a reasonably close cousin of the `remapTo()` we encountered earlier; you could read it as "as if it were the direct object for" so that `dobjFor(Search) asDobjFor(LookIn)` means "when the nest is the direct object of a **search** command treat it as if it were the direct object of a **look in** command", or "treat **search nest** as equivalent to **look in nest**".

Up to this point we've only used `dobjFor()` to make one command behave like another, but obviously there has to be more to writing a game than that: at some point we have to define what a command actually *does*, and this is what the example above does for `dobjFor(LookIn)`. Note first of all that when we do that we follow `dobjFor()` with a block of code enclosed by braces {}. Within that block we have what looks like a method called `action()` with its code enclosed in a further set of braces {}. This may look rather strange; it is in fact exactly equivalent to defining a method called `actionDobjLookIn()` (without `dobjFor()` and the outer enclosing braces), and you could indeed do it that way. We tend to use `dobjFor()` instead because it makes the code easier to read and write.

We'll give a fuller account of all this at the end of the chapter, so don't worry if it still seems a little hazy at the moment. The main thing to note right now is that the method that looks like it's called `action()`, what we might call the action part of the dobjFor() block, is the place where we put the code that defines what actually happens when the nest is looked in.

There are four further points to note about this code: (1) `moved` is a property defined by the library; it starts at `nil` (i.e. not true) and is set to `true` as soon as the object is moved from its initial location, which is what `ring.moveInto(self)` does; (2) to move the ring we use its `moveInto` method, we do *not* change its `location` property directly by writing something like `ring.location = self` (part of the reason for this is that the library maintains lists of what's contained by what; the `moveInto` method takes care of updating the appropriate lists when an object is moved, while setting the location property would not); (3) `self` simply refers to the object in which it occurs (in this case the nest); (4) `exit` terminates the action straightaway, so that code following an `exit` statement is not executed; the `exit` statement may also be used to veto an action at the check stage, as we shall see shortly.

An alternative way of achieving the same effect would be to leave the +++ in front of the definition of ring, and add `PresentLater` to the front of its class list, at the same time changing `ring.moveInto(self)` to `ring.makePresent()` in the definition of `nest`. Where the `PresentLater` mix-in class is used,[20] the game initialization makes a note of the object's location, then moves it into `nil` (i.e. out of the game world); a call to `makePresent()` then restores the object to its initial location. Yet another way we could achieve much the same effect would be by making the ring of class `Hidden`, but we shall illustrate that on another object shortly.

[20] A mix-in class is a class that must be used, or 'mixed in' with another; so, for example, in this case we should start the definition of ring with `ring : PresentLater, Thing` and not just `ring : PresentLater`.

The revision to the nest object makes things slightly more interesting, but searching the nest isn't much of a challenge. It would be rather more interesting if in order to search the nest we first had to hold it, and, furthermore, if the nest was just out of reach so we first had to find some way of bringing it nearer. The obvious tool for the job would be some sort of stick, and the obvious place to find such a stick might be among twigs and branches lying at the foot of the tree.

But first we must prevent Heidi from looking in the nest until she's holding it. To do that we can use `check()`. Like `action()`, `check()` is a method that goes inside the braces following `dobjFor()`, but whereas we use `action()` to define what happens when the command is carried out, we can use `check()` to stop it being carried out and explain why. In this case we don't want to stop it unconditionally, but only if the nest isn't held:

```
check()
    {
      if(!isHeldBy(gActor))
      {
        "You really need to hold the nest to take a good look at
          what's inside. ";
        exit;
      }
    }
```

Our next job is to make it impossible to take the nest without the use of the stick. TADS 3 already defines what happens when the player tries to take something, so what we need to do here is to change that behaviour; we do this by overriding the nest's `dobjFor(Take)` routines. Nonetheless, when we do allow the action to go ahead, we'll still want the normal library handling to work; to ensure that happens we need to include `inherited` in the action part, which calls the action handling for **take** that the nest inherits from its superclass. To ease the player's "guess the verb" hassle we'll let the player character take the nest if she's simply carrying the stick. The appropriate code, which illustrates both a check and an action section in the same `dobjFor()` block, is as follows:

```
dobjFor(Take)
  {
    check()
    {
      if(!moved && !stick.isIn(gActor))
      {
        "The nest is too far away for you to reach. ";
        exit;
      }
    }
    action()
    {
      if(!moved)
        "Using the stick you manage to pull the nest near enough to take,
          which you promptly do. ";
      inherited;
    }
  }
```

We include the `if(!moved)` condition[21] in both `check()` and `action()` here on the assumption that once the nest has been moved, it won't be put back out of reach.

[21] Note that ! is the negation operator, so that if(!moved) is true if moved is nil.

The `inherited` statement at the end of the action method ensures that we actually do end up taking the nest (by continuing with the standard behaviour for the take action). But this raises another issue: inherited() means roughly "at this point carry out what would have happened if we hadn't overridden this method", but what would have happened is that not only would Heidi have picked up the nest, but this would have been reported with the laconic response "Taken." That's fine, except the first time we take the nest (when `moved` is still `nil`), when we don't want to see "Taken" in addition to our custom message about moving the stick. Actually, this won't be a problem; if you try running this code you'll find that the "Taken" message doesn't appear alongside the other message. That's because the library doesn't use a double-quoted string to produce it, but a macro called `defaultReport()`, in this case. `defaultReport('Taken. ')`, and a defaultReport is only shown if there's no other report.

This is fine, but it might not occur to the player that the nest can be taken simply because Heidi is holding the stick; the player may suppose other command needs to be used to bring the stick nearer, such as **move nest with stick**, so we'll code this command to act in exactly the same way. This introduces a new complication, defining special behaviour for a verb involving two objects. We'll begin by defining some code on the intended direct object of this command, which is still the nest:

```
dobjFor(MoveWith)
  {
    verify()
     {
       if(isHeldBy(gActor))
          illogicalAlready('{You/he} {is} already holding it. ');
     }
    check()
    {
      if(gIobj != stick)
        {
          "{You/he} can't move the nest with that. ";
          exit;
        }
    }
  }
```

The effect of this code is to rule that it is illogical to attempt to move the nest with anything while the nest is being held, and not to allow the nest to be moved with anything other than the stick.

This code snippet introduces several new features we should pause to consider. First, just as in TADS 3 the abbreviation dobj will normally refer to a direct object, so iobj will normally refer to an **i**ndirect **obj**ect, the second object involved in a two-object command like **move nest with stick**. But in the check routine above what we actually see is something call `gIobj`. Here the g effectively stands for 'global'; although TADS 3 does not in fact support global variables, it has several things that look and act like global variables, and `gIobj` is one of them (in fact these pseudo-global variables are shorthand ways of referring to the properties of objects in which the values are actually stored, but that needn't concern us now). The common not-exactly-global variables you'll often encounter in TADS 3 include: `gDobj` (the direct object of the current action), `gIobj` (the indirect object of the current action), `gAction` (the current action), `gActor` (the actor performing the current action, which is usually but not necessarily the player character), and `gPlayerChar` (the object representing the player character).

The next new feature we've introduced in this code is a `verify()` routine. We'll be giving the full story on `verify()` shortly, but the very brief version is that, like check(), verify can be used to veto an action, but that unlike `check()`, `verify()` affects disambiguation (which object the parser thinks a player's command most probably refers to). A verify() routine can contribute to disambiguation without vetoing an action, but when it does disallow an action it (nearly) always does so with statement of the form `illogical('Reason why this action is plain daft. ')`. `illogicalAlready()` is just a specialized form of `illogical()`, used when the action is pointless because it's trying to bring about something that's already the case (for example, trying to open a door that's already open).

The third new feature is the use of parameter substitution strings, meaning those strange things in curly braces: `{you/he}` and `{is}`. The point of these is that when they are actually displayed in a game, they are replaced with the appropriate text. So, for example, if the player issues the command **move nest with stick** '`{you/he} {is}`' becomes 'you are', but if an NPC called Fred attempted the action it would become 'Fred is'. For the full story on parameter substitution strings, read the 'Message Parameter Substitutions' article in the Technical Manual.

The next job is to make the stick handle its part of the action. You'll recall that the stick is the *indirect* object of this command, so you might guess that just as we need to define `dobjFor()` on the direct object, we need to define `iobjFor()` on the indirect object. If you did guess that, you'd be right:

```
iobjFor(MoveWith)
  {
    verify() {}
    check() {}
    action()
    {
      if(gDobj==nest && !nest.moved)
        replaceAction(Take, nest);
    }
  }
```

The first thing to note is the `verify()` and `check()` routines that do nothing. We need them to do nothing to make sure they don't veto the use of the stick as the indirect object of a **move with** command. Without that empty `verify()` method **move nest with stick** would produce the response 'You cannot move anything with the stick.' We don't actually need to provide an empty `check()` method here, since the library version is already empty, but including it here does no harm, and if we didn't *know* that the library didn't rule out **move with** in `check()` it would be as well to include it just to make sure.

The `action()` routine only does anything if the direct object (`gDobj`) is the nest and the nest hasn't already been moved. If either of those conditions fails, the command will result in the default MoveWith action of the direct object, which simply reports that moving the direct object didn't achieve anything. If, however, the direct object is the nest and it hasn't been moved yet we want the result to be the same as if we had issued the command **take nest**. We achieve this with the `replaceAction` macro, which does just what it says it does and stops processing the current command, replacing it with the new command. Had we wished to execute another command and then continue with the existing command we would have used `nestedAction` instead.

You may be wondering how `replaceAction()` differs from `remap()`, which we used before. The main difference is that remapping happens before anything else (in particular, before `verify()` and `check()`), while `replaceAction()`, which can really only go in the `action()` part, happens *after* `verify()` and `check()`; `replaceAction()` and `nestedAction()` are also more flexible than `remap` in that they can be combined with other code in the `action()` routine; we can do other things before replacing the action, but we can't do anything before remapping.[22]

You may also be wondering why we put the code for taking the nest in the `action()` routine of the indirect object rather than the direct object here. The answer is, there's no reason at all for doing it that way, apart from illustrating the use of an `action()` routine on an indirect object (and incidentally taking advantage of the fact that the indirect object's `action()` routine generally runs before the direct object's `action()` routine). Other than that, it would have been just as good, if not better, to have put this handling on the direct object, thus:

```
dobjFor(MoveWith)
{
    verify()
     {
       if(isHeldBy(gActor))
          illogicalAlready('{You/he} {is} already holding it. ');
     }
    check()
    {
      if(gIobj != stick)
        {
          "{You/he} can't move the nest with that. ";
          exit;
        }
    }
    action()
    {
       if(!moved)
          replaceAction(Take, self);
       else
          inherited();
    }
}
```

Indeed, this would probably have been a clearer and neater way of doing it. In this revised version, note the use of `inherited()` to call the base class handling of MoveWith once the nest has been moved. Note also that `else` on the line before is not strictly necessary since once `replaceAction()` is executed the original (MoveWith) action is stopped in any case.

More broadly, this illustrates that the action() part of a two-object command can be written on either the direct object or the indirect object or even split between both, since both action routines will be executed (unless `exit` or one of its close relatives is used to break out of them), with the indirect object's action handling usually (but not necessarily) called before the direct object's. So how do we decide which to use? One rule of thumb might be to write the action() handling on the object most affected by the action. Another might be to write it on the object which most affects the outcome of the item: for example, if your game contains a knife, a sword and a laser rifle, and cutting things with each of these produces greatly different results, you might want to write the CutWith action handling on the indirect object, whereas if it doesn't make much difference what you cut with but a great deal of

[22] A further distinction, which is a complication too far to go into here, is that remapping can take place before object resolution is completed. In particular if we remap on one object of a two-object command we can't assume that the other object has yet been determined.

difference whether it's a piece of solid rock, a window, a lump of butter, or Aunt Gertrude that you're cutting, you'll probably be better off putting the action handling on the direct object. A third rule of thumb might be to put the action handling on the object that exhibits the most exceptional behaviour; so, for example, most objects won't allow things to be put inside them, but Containers will, so the library action handling for **put x in y** goes on the indirect object (the Container), not the thing that's being put in the Container. Other cases may be less clear-cut (no pun intended), in which case it's best to choose one or the other and stick with it consistently (having an action like CutWith handled on some direct objects and some indirect objects is a likely recipe for confusion).

Enough of that digression; let's return to the stick. Although the base of the tree is a good place to find the stick, it's probably better not to make it too obvious; if the stick is just lying there in plain sight the player will take it automatically, which will make getting hold of the nest virtually a non-puzzle. We'll make things harder by burying the stick in a pile of useless twigs, so that the player has to do some work to find them. While we're at it we'll change the description of the sycamore tree so that it refers to the pile of twigs. Again, this is something you might like to try yourself before reading on to see how this guide does it. After changing the description of the tree, you'll need to add one object to represent the pile of twigs, and then another for the stick object, which should remain hidden until Heidi examines or searches the pile of twigs. To hide the stick you could use one of the techniques discussed in relation to hiding the ring in the nest, or you could make the stick a `Hidden` object and call its `discovered()` method at the appropriate moment.

Here's one way of doing it:-

```
+ tree : Fixture 'tall sycamore tree' 'tree'
        "Standing proud in the middle of the clearing, the stout
      tree looks like it should be easy to climb. A small pile of loose
      twigs has gathered at its base. "
    dobjFor(Climb) remapTo(Up)

;

+ Decoration 'loose small pile/twigs' 'pile of twigs'
  "There are several small twigs here, most of them small, insubstantial,
   and frankly of no use to anyone bigger than a blue-tit <<stick.moved ?
   nil : '; but there is also one fairly long, substantial stick among
    them'>>. <<stick.discover>>"
  dobjFor(Search) asDobjFor(Examine)
;

+ stick : Hidden 'long substantial stick' 'long stick'
  "It's about two feet long, a quarter of an inch in diameter, and
    reasonably straight. "
   iobjFor(MoveWith)
  {
    verify() {}
    check() {}
    action()
    {
      if(gDobj==nest && !nest.moved)
        replaceAction(Take, nest);
    }
  }
;
```

We could have handled the stick in a similar manner to the ring, by moving it into the clearing when we wanted it to appear, but this seems a good opportunity to introduce the `Hidden` class, which does much what it says. A `Hidden` item is one that is physically present but does not reveal its presence until its `discover` method is called. By making `stick` of class `Hidden` instead of class `Thing`, we can control when we want it to appear. In this case we want it to appear when the player character examines the pile of twigs, so we make an embedded call to `stick.discover` in the description of the twigs, using the <<>> syntax. The fact that this method will be called every time the twigs are examined doesn't matter, since once the method has been called once, the subsequent calls will have no effect. There are a couple of other refinements we need to think about, however. First, the description of the pile of twigs should only refer to the stick amongst them until the stick has been moved; we achieve this through another embedded expression, `<<stick.moved ? nil : '; but there is also one fairly long, substantial stick among them'>>`, that displays nothing if the stick has been moved but describes the stick if it hasn't. Secondly, the player might reasonably try to **search** the pile of sticks as well as **examine** them, so we add a line to the definition of the anonymous sticks object to remap **search** to **examine**.

Note how we have defined the `vocabWords` property of the `Decoration` object representing the pile of twigs: we have defined it as `'loose small pile/twigs'`. Although you can't normally refer to an object by two or more of its nouns, there is an exception in the case of a name like 'x of y', where both x and y should be specified as nouns. Our pile of twigs will now respond to **examine twigs** or **x small pile** or **search pile of loose twigs** and other such combinations.

Let's just add one final refinement. Normally if you **drop** an object, it lands in the room where you are, as you would expect. But if you were to drop something from the top of a tree you'd expect it to fall to the ground below rather than hover around in the air still conveniently in reach. It would be good if we could model this in our game, and it turns out to be fairly straightforward. First, we need to change the class of `topOfTree` to `FloorlessRoom`, which means that any object dropped or thrown from this location won't land here. Then we need to override `topOfTree's bottomRoom` property to define where something dropped from there *will* land. In this case we want `bottomRoom` to be `clearing`. Now anything dropped (or thrown) while Heidi is at the top of the tree will fall to the clearing, and the game will display a suitable message to show that the object is falling out of sight. The definition of `topOfTree` thus becomes;

```
topOfTree : FloorlessRoom 'At the top of the tree'
         "You cling precariously to the trunk, next to a firm, narrow
            branch. "
         down = clearing
         enteringRoom(traveler)
         {
           if(!traveler.hasSeen(self) && traveler == gPlayerChar)
             addToScore(1, 'reaching the top of the tree. ');
         }
         bottomRoom = clearing
;
```

In the next chapter we shall learn how the ring came to be in the nest, who it belongs to, and how to win the game. To do that we shall need to create a Non Player Character (NPC), and that will be our central task.

5. *Controlling the Action*

The TADS 3 library defines a fair number of actions (see the end of the en_us.t library file for their grammar definitions, which will give you some idea of the range of actions available), together with default responses. For some standard actions like TAKE, DROP, EXAMINE, OPEN, LOCK and PUT ON the standard responses are often all you need; for many of the others, such as BREAK, CLEAN, JUMP OVER or POUR the standard response is merely a message saying that the proposed action is impossible or that it has no effect.

As we have seen over the last few pages, you won't get very far in writing a game in TADS 3 without customizing the standard library behaviour for various actions under particular circumstances. Indeed, a substantial part of writing IF-code consists in just this task (writing custom action handling). Over the course of this chapter we've covered quite a few of the concepts and methods used to customize actions in TADS 3, but before we carry on to anything else, we should first review what we have learned in a more systematic fashion, filling in all the most significant gaps as we go.

So, to start again from the beginning. Customizing actions in TADS 3 generally involves using the `dobjFor()` and `iobjFor()` constructs. Wherever you see dobj and iobj in TADS 3, remember that they are abbreviations for **d**irect **obj**ect and **i**ndirect **obj**ect. IF typically has three kinds of commands: simple commands like **look** and **inventory** that have no objects at all, single-object commands like **climb tree** or **take stick** that have one object, called the direct object, and two-object commands like **move nest with stick** or **put nest on branch** or **hit troll with sword** that have both a direct object (in these examples the nest or the troll) and an indirect object (in these examples the stick, the branch or the sword). The direct object is normally the one that immediately follows the verb, while the indirect object is usually the second of the two objects, and usually comes after a preposition such as 'with' or 'on'. I say 'usually' because English sometimes allows more than one phrasing: **give fred the book** means the same as **give the book to fred**, and in both cases the book would be the direct object and Fred would be the indirect object. If in doubt, always think of the longer version of the command phrasing that includes the preposition (such as 'to') when deciding which object is the direct object and which the indirect object.

Customizing the behaviour of a single-object command, like **enter the cottage** or **climb the tree** is relatively straightforward. You simply need to define `dobjFor(Whatever)` on the direct object of the command and then specify what happens (how exactly we do that is something we'll get to in a moment). For a two-object command such as **move nest with stick** it's more complicated; you often need to define `dobjFor(Whatever)` on the direct object and `iobjFor(Whatever)` on the indirect object. At the very least, if you want the action to go ahead, you need to ensure that both of the objects involved in the action will allow it, so that the play doesn't get a message like "You can't move the nest" or "You can't move anything with the stick".

A further complication, which we won't go into very far here, is that there can be things that look like direct or indirect objects but are considered to be something else by TADS 3. For example in the command **ask fred about the weather**, Fred is indeed the direct object, but 'the weather' is the *topic object*. In the command **enter qwerty on keyboard**, the keyboard is actually the direct object and 'qwerty' is the *literal object*. Beyond making sure that you're aware of this complication, we can afford to ignore it for now (when you want the full story, read the article on How to Create Verbs in the Technical Manual).

Before considering how to use the `iobjFor()` and `dobjFor()` macros, it may be a good idea to take a closer look at what they actually mean. They are, in fact, macros that define *propertysets*, which are simply a short-hand device for defining a set of properties which have a common element in their name (e.g. fooTake, fooDrop and fooBar, all of which start with foo). In the case of the dobjFor and iobjFor macros, it's the name of the action (e.g. Take) plus the role of the action (dobj or iobj) that's the common element. So if you write:

```
dobjFor(Take)
{
   foo = 'poop'
   bar() { say(foo); }
}
```

This is exactly the same, so far as the compiler is concerned, as if you had written:

```
fooDobjTake = 'poop'
barDobjTake() { say(foo); }
```

The above example is not especially useful, since the library makes no use of these property and method names (although you could always define them to do something useful in your own code); the library does, however, call a number of properties and methods on the direction object and indirect object of any action. For example, if the action is TakeWith the following properties/methods will be invoked respectively on the direct and indirect objects of the command:

```
remapDobjTakeWith                 remapIobjTakeWith
preCondDobjTakeWith               preCondIobjTakeWith
verifyDobjTakeWith()              verifyIobjTakeWith()
checkDobjTakeWith()               checkIobjTakeWith()
actionDobjTakeWith()              actionIobjTakeWith()
```

Any of these properties/methods may be defined (or invoked) using these names (and sometimes it may be useful to do so; dobjFor and iobjFor merely provide a convenient way of defining these properties without having to remember their full names, and for grouping the related methods together in the code layout; e.g.

```
dobjFor(TakeWith)
{
   remap = nil
   preCond = [touchObj]
   verify()
   {
       if(heldBy == gActor)
         illogicalAlready('{You/he} {is} already holding {the dobj/him}. ');
   }

   check()
   {
     if(gDobj == poisonousSnake)
       failCheck('You think better of it. ');
   }
   action()
   {
      "{You/he} take{s} {the dobj/him} with {the iobj/him}. ";
      gDobj.moveInto(gActor);
   }
}
```

The verify(), check(), action() and preCondition methods are described in some detail in the articles "How to Create Verbs in TADS 3", "TADS 3 Action results" and "On good usage of verify() and check() in TADS 3 games" in the *Technical Manual*. These are all articles you will want to read sooner rather than later; you might find it particularly useful to read the "TADS 3 Actions results" article at the end of this chapter if you want more help on the ground we're about to cover.

a. *Verify()*

The verify method has two main functions: (a) to veto actions that plainly should not be allowed to continue (e.g. EAT MOUNTAIN) and explain the veto, and (b) to help the parser decide which object to choose in the case of ambiguity (e.g. if OPEN DOOR could refer to either the currently open red door or the currently closed blue door, verify can be used to prefer the blue door, since you can't open a door that's already open).

A verify() routine should never alter the game state (since, among other things, it will probably be called multiple times during object resolution).[23] Neither should it directly display a string using a double-quoted string or `say()`. Normally it should only contain one or more of the macros designed to be used in a verify routine, if-statements to determine which macro to use, or the `inherited` keyword to invoke a superclass's verify behaviour. A verify routine can also simply be empty (i.e. contain no code at all); this is often useful when you want to allow an action to proceed unconditionally.

The macros that can be used in verify routines to define verify results are:

```
logical
logicalRank(rank, key)
logicalRankOrd(rank, key, ord)
dangerous
illogicalAlready(msg, params…)
illogicalNow(msg, params…)
illogical(msg, params…)
illogicalSelf(msg, params…)
nonObvious
inaccessible(msg, params…)
```

Of these, only the five that start with i (illogical or inaccessible) will prevent an action altogether, the rest mainly make the object more or less likely to be chosen as the object of the action by the parser in case of uncertainty. This is why it is only these five i-macros that take a msg parameter, which is single-quoted string (or property returning one) explaining why the action may not proceed. For now, it's only the four `illogical` macros that you need to worry about (`inaccessible` is rarely

[23] There are minor exceptions to this; it is, for example, perfectly legitimate to include a <.reveal> tag in a message displayed from a verify routine, which might be useful for your hints system, but such exceptions are beyond the scope of this introductory guide, and for now it’s best to stick rigidly to the rule.

needed in game code). Each of the four will prevent an action from continuing, but in case of ambiguity the parser will choose an object that returns an `illogicalAlready` result in preference to one that returns an `illogicalNow` result, and either to one that returns a plain `illogical` result. All three will be preferred to something that returns `illogicalSelf`.

For example, suppose your game has a red door, a red box and a red cup. It's perfectly logical to open a door, but it's not a good choice for an OPEN command if it's already open. Likewise, a box can probably be opened, but perhaps this red box can be broken, and once broken, it's no longer openable. Finally, a cup is never something that can be opened; the command OPEN CUP would never make sense. You might define the corresponding `verify` methods as follows:

```
redDoor: Door 'red door*doors' 'red door'
  dobjFor(Open)
  {
   verify()
   {
     if(isOpen)
        illogicalAlready('The red door is already open. );
   }
  }
;

redBox: OpenableContainer 'red box*boxes' 'red box'
  dobjFor(Open)
  {
     verify()
     {
         if(isBroken)
            illogicalNow('You can no longer open it; it\'s broken. ');
     }
  }
;

redCup: Container 'red cup*cups' 'red cup'
   dobjFor(Open)
   {
       illogical('You can\'t open a cup. ');
   }
;
```

If you type **open red** when all three red objects are present, then if the red door is not closed and the red box is not broken, the parser will ask which red object you mean (since under these circumstances either would be equally logical). If the red door were already open, but the box not yet broken, the parser would choose the red box; if the red door were already open and the box broken, the parser would choose the red door (and report that it's already open). Finally, if the player character took the broken red box and the red cup to another location and you then typed **open red**, the parser would choose the red box (and report that you can't open it because it's broken).[24]

Note that in the library the argument to these illogical macros is typically a property pointer (e.g. `&cannotTakeMsg`); roughly speaking, these refer to properties of the playerActionMessages object (defined in msg_neu.t). But this is a complication

[24] Obviously this example is a bit oversimplified; in a real game it would also be necessary to test whether the box was already open, and apply an illogicalAlready macro if it was.

we shall set to one side till later; in your own code, at least to start with, you can stick to using single-quoted strings.[25]

The non-i macros all allow the action to proceed (at least to the check stage, see below), but again affect how likely the object is to be chosen by the parser in case of ambiguity. The `logical` macro is simply the default; a `verify()` routine that consists solely of the keyword `logical` is identical to one that contains nothing at all. The `logical` macro is thus strictly speaking redundant, but it may improve the readability of code to use it in a complex `verify()` routine with several different conditions producing different results.

The `logicalRank(rank, key)` macro allows you to choose the priority the parser gives to selecting different objects in cases of ambiguity. The default rank is 100; an object with a logical rank of 150 is regarded as a particularly suitable target for a command, while one with a logical rank of 50 would be a possible but not very likely one.

Suppose that in the previous example, when the the door is closed and the box not broken we wanted the parser to prefer the door to the box in response to an **open red** command.. To boost the ranking of the door, we might use `logicalRank` thus:

```
redDoor: Door 'red door*doors' 'red door'
  dobjFor(Open)
  {
   verify()
   {
     if(isOpen)
        illogicalAlready('The red door is already open. );
     else
        logicalRank(120, 'door');
   }
  }
;
```

Since the (unbroken) red box has a default logical rank of 100, **open red** will now prefer the door. Note that in this example, `'door'`, the second parameter to the `logicalRank` macro, is the key value; this is effectively an arbitrary single-quoted string. Technically it can be used by the parser in breaking a tie (if two objects have the same logicalRank with the same key, then the parser knows that it can ignore this and look for some other way of breaking the tie), but in practice it seldom matters in game code what you put here.

The final two macros (`dangerous` and `nonObvious`) allow an action to proceed, but only if the player unambiguously names the object. Both macros also prevent the object ever being chosen as a default object for the command in question, or in an implicit action.

The `dangerous` macro is intended to prevent an object being used in an action when carrying out the action on that object would be plainly dangerous (e.g. **drink poison**) even though the action is perfectly possible and the object might be the only suitable one in scope. Thus, to prevent a bare **drink** command from making the PC drink the poison when the poison is the only potable object around, you would define the poison's `verifyDobjDrink()` method as `dangerous`.

[25] Although once you gain more experience with TADS 3 it can be useful to define your own playerActionMessages properties and override the library ones. One advantage of defining your own is to be able to use the same custom message (e.g. 'That would be highly uncomfortable. ') from multiple objects without having to define the same text each time. See below.

```
poison: Thing 'deadly poison' 'deadly poison'
   dobjFor(Drink)
  {
      verify() {  dangerous; }
      action()
      {
            "You feel an unpleasant choking sensation as the poison
             burns down your throat; then you feel no more. ";
            finishGameMsg(ftDeath); // this ends the game
      }
  }
;
```

The `nonObvious` macro works similarly, but makes the object so marked even less likely to be chosen in the event of ambiguity. Its main purpose is to prevent a puzzle being solved accidentally by having an action carried out implicitly or by default. For example, if we hadn't wanted to make it obvious that the nest could be moved with the stick, we could have put `nonObvious` in the `verify()` section of the `iobjFor(MoveWith)` on the stick.

One last point: the macros we have just been discussing (`illogical` and the like) are for use *only* in verify() routines.

So far we have discussed `verify()` mainly in terms of how the parser selects objects in case of ambiguity. Another way of looking at is that `verify()` should be used to veto an action only if it should be obvious to the player (not the player character) that the action is illogical (e.g. eating a mountain or opening an already-open door). In fact, these two ways of looking it amount to the same thing: the purpose of `verify()` is to help the parser decide what the player probably meant in case of ambiguity. If you want to veto an action which is perfectly 'logical' (i.e. one that the player could well have meant) you should therefore use `check()` instead.

b. *Check()*

The purpose of a `check()` routine is to veto an action that is perfectly logical, but should not be carried out for some other reason. As with `verify()`, `check()` should not be used to change the game state.[26] All that a `check()` routine should do is either allow an action to go ahead, or else forbid it by displaying a message and using the `exit` macro.

For example, suppose we gave Heidi a dress. Removing the dress would be a perfectly logical action, and so the dress would be a good choice for an ambiguous Doff action,. We don't want to make removing the dress illogical, since it may well be what the player intends. On the other hand, having Heidi remove her dress in the course of her adventures may seem rather out of character, and it would serve no useful purpose in the game, so we probably want to prevent it. The best place to do this would be in a `check` routine:

```
dobjFor(Doff)
{
   check()
   {
       reportFailure('You can\'t wander around half naked! ');
       exit;
```

[26] Except in some trivial way such as setting a flag to show that the check routine has disallowed an action, which might be useful, for example, when constructing a hint system.

```
   }
}
```

Note that here we have used the `reportFailure` macro; it's not strictly necessary to do so here: we could have used a double-quoted string to display the text, and it would have worked just as well. However, using `reportFailure` is a good habit to get into, since in other contexts (outside a `check()` routine) it can be used to signal that an action failed, which can sometimes produce better implicit action reports (e.g. '(first trying to open the door)' rather than '(first opening the door)' when the attempt to open the door fails).

The use of `reportFailure` followed by `exit` is so common in `check()` routines that Thing defines a `failCheck()` method that combines them both into one statement. The foregoing example could then be written:

```
dobjFor(Doff)
{
   check()
   {
       failCheck('You can\'t wander around half naked! ');
   }
}
```

And our dress object (located immediately after the me object in the code) could then be defined as:

```
+ Wearable 'plain pretty blue dress' 'blue dress'
  "It's quite plain, but you think it pretty enough. "
  wornBy = me
  dobjFor(Doff)
  {
     check()
     {
       failCheck('You can\'t wander around half naked! ');
     }
  }
;
```

Of course, there's no reason why an action should not fail in the `action()` routine as well in `check()` – the `failCheck()` method would work perfectly well in `action()`, so you may wonder what real purpose `check()` actually performs, beyond making the code look a bit neater if failure is conditional. There are in fact three main reasons for separating `check()` from `action()`:

- As we have seen, a good deal of TADS 3 programming consists in overriding the library's standard verb handling. It is often convenient to override check() separately from `action()` if we want to change only the conditions under which the action isn't allowed to happen, or only what happens when the action does happen.
- In a two-object command (such as **put bag on table**) the `action()` routines are called on both the indirect object and the direct object. Ruling out the action in check() ensures that the action is stopped before either `action()` routine is called (otherwise we might find, for example, that the indirect object's `action()` method carried out the action before the direct object's `action()` method could stop it).

- From version 3.0.15.1 onwards the `check()` routines can be made to run before action notifications such as `beforeAction()`. For this to work you need to set `gameMain.beforeRunsBeforeCheck` to `nil`. We'll explore this a bit further below.

If an action is vetoed (either by check, verify or preCond) before it reaches the action stage, no action notifications will be sent, and so nothing that might have reacted to the action in a `beforeAction` or `afterAction` method will in fact do so. But note, *`check()` only runs before `beforeAction()` if you have set `gameMain.beforeRunsBeforeCheck` to nil*. In some future release of TADS 3 (perhaps 3.1) this may become the default, but for now you have to set this option if you want it (this ensures compatibility with games written prior to version 3.0.15.1). This is what we did when we first defined the `gameMain` object back on page 44, and we suggest you always do too.

For example, suppose we went on to define an NPC who reacted to Heidi undressing herself in front of him, with something like:

```
afterAction()
{
  if(gActionIs(Doff) && gActor==gPlayerChar && gDobj==dress)
     "<q>Hey! What do you think you're doing, young lady!</q> cries
     the charcoal burner. ";
}
```

If we had simply displayed the message about not wandering around half-naked in the action() routine, we might end up with a transcript like this:

>**remove dress**
You can't wander around half naked!

"Hey, what do you think you're doing, young lady!" cries the charcoal burner.

By vetoing the **doff** action in the check() routine, we ensure that the charcoal burner never gets a chance to react, and so we won't get his inappropriate response to an action that is not, in fact, carried out.

c. *Action()*

The action() routine is in a sense the most straightforward to understand, it's the routine that does the actual work of carrying out an action (once it's passed the verify, precondition and check stages). However, depending on the nature of the action, it may contain the most complex code, since it's here that the game state may actually be changed. We have already seen several examples of `action()` routines, most recently that for drinking the poison on p. 75 above. The only complication to bear in mind is that if you define `action()` routines on both the direct and indirect objects of an action, both action routines will be carried out (the indirect object's first), so you need to make sure that their combination does what you want; normally, it's better to define an action routine on one or the other of the objects involved in a two-object command, but not both.

d. *PreCond()*

There are certain necessary conditions that tend to recur commonly in IF actions. In order to read the book or examine the chest, the objects in question have to be visible. In order to hit the troll with the sword, or move the nest with the stick, or unlock the chest with the gold key, the sword, stick or key first have to be held. These common conditions are encapsulated in TADS 3 as `PreCondition` objects, since they represent the common preconditions of various types of a command.

The `preCond` property in `dobjFor` or `iobjFor` propertyset contains (or, if it is a routine, returns) a list of the preconditions needed for the object in question to be used in the action in question. For example, in the case of reading the book and examining the chest you might define:

```
        preCond = [objVisible]
```

Whereas, in the case of hitting the troll or whatever, in the appropriate `iobjFor()` section (for AttackWith, MoveWith, UnlockWith) you might define:

```
        preCond = [objHeld]
```

The library already defines sensible default preconditions for most actions, so you often don't need to worry about them. However, as your own games become more sophisticated, you may want to adjust the preconditions that apply to a particular action on a particular object. For example, you may decide that a particular book needs to be held as well as being visible in order to be read, so you might define:

```
redBook: Readable 'little red book*books'   'red book'
…
        dobjFor(Read)
        {
           preCond = [objVisible, objHeld]
           …
        }
;
```

Like check() and verify() routines, preconditions can veto an action if certain conditions aren't met. Unlike check() and verify(), however, they can be used to change the game state to meet the precondition. For example, the `objHeld` precondition will first check to see if the object is already held; if it isn't, it will try to make the actor take the object (via an implicit action, i.e. one that is reported as '(first taking the book)' or whatever). If the implicit action succeeds, the action is allowed to proceed (provided there's nothing else preventing it). If not, the action is disallowed, with a message like '(first trying to take the book)' that explains the reason for the failure (e.g. "The book is out of reach."). Some preconditions, however simply test whether the condition is met, and disallow the action if it isn't. For example, an `objVisible` precondition makes no attempt to make an object visible if it isn't (in the general case it's impossible to know what implicit action, if any, could bring this about), it simply vetoes the action if the object can't be seen by the actor.

The library defines several preconditions, including `objOpen`, `objClosed`, `objUnlocked`, `touchObj`, `actorStanding`, `objAudible`. For a complete list, see the pre-conditions section of the "TADS 3 Action Results" section of the *Technical Manual*. It is also perfectly possible (and often useful) to define your own, although that is beyond the scope of this guide. The best way to get a full understanding of preconditions is to study the library file precond.t.

e. *Remap()*

We have already encountered remapping via the `remapTo` macro; it's used when we want to remap a command on one object to a different command involving the same or maybe different objects (or even no objects at all). So, for example, on the tree object we earlier defined:

```
dobjFor(Climb) remapTo(Up)
```

What such code actually does is to make the appropriate remap property return a list containing the action followed by the objects involved in the action; the example above is in fact equivalent to:

```
remapDobjClimb = [UpAction]
```

A more complicated example was the cottage, where we defined:

```
dobjFor(Enter) remapTo(TravelVia, insideCottage)
```

which is equivalent to:

```
remapDobjEnter = [TravelViaAction, insideCottage]
```

Mostly, you can use the `remapTo` macro without worrying about the underlying code the compiler actually sees, but there are a couple of cases where understanding the underlying code can be important. The first thing to realize is that if there is a remap in operation (that is the remap property is non-nil) this will take precedence over all the other action properties (preCond, verify, check and action). In some cases this can lead to unexpected results: you may define verify, check and action for some verb on some object, but find that the object is doing something quite different from what you defined. The reason may very well be that your object has inherited a remap from one of its superclasses, and that this remap is taking precedence over your customizations of the other methods.

For example, the library Room class remaps LookIn to Examine. Normally this is a perfectly sensible interpretation, but you might decided that's not what you want in your game. For example, you might feel that a player who types **search room** or **look in room** is just being lazy, and should be instructed to concentrate on the objects within the room instead. So you might write:

```
modify Room
  dobjFor(LookIn)
  {
    verify()
    {
     illogical('Try examining some of the objects in the room instead. ');
    }
  }
;
```

But you'd find that this didn't actually change anything; **look in room** and **search room** would still result in the room being examined, since the remap would still be in action. What you'd actually need to do is to reset the remap to nil:

```
modify Room
  dobjFor(LookIn)
  {
    remap = nil
    verify()
    {
     illogical('Try examining some of the objects in the room instead. ');
    }
  }
;
```

The second area where it can be useful to know how the underlying code works is with conditional remapping. The library defines a macro called `maybeRemapTo()`, which only remaps if a certain condition holds (that is, if its first parameter is true). For example, if you had a gate object, and you wanted **push gate** to be treated as **close gate** when the gate was open, but in the normal way when the gate was closed, you could define:

```
gate: Door 'gate'  'gate'
  dobjFor(Push) maybeRemapTo(isOpen, Close, self)
;
```

The underlying code here is in fact:

```
remapDobjPush = (isOpen ? [CloseAction, self] : inherited());
```

The first thing to note is that if the condition is not met (in this case, if the gate is not open), what one gets is not necessarily no remapping, but the inherited remapping. Often the inherited remapping is in fact `nil`, but it might not be. For example, suppose you had a room in your game called the Study, and you gave it a `vocabWords` property of 'study', so that it could be the taget of commands. In particular you decide that you want the player to be able to enter the command **search study**, and have this command find an important letter lying behind the curtain (or wherever), if the letter hasn't already been found. You're aware that `Room` inherits `dobjFor(Search) asDobjFor(LookIn)` from `Thing`, so you decide to implement finding the letter in `dobjFor(LookIn)`. If, however, the letter has already been discovered, you decide you want **search study** or **look in study** simply to perform a **look** command, so you write:

```
dobjFor(LookIn) maybeRemapTo(letter.discovered, Look)
```

Unfortunately, this wouldn't work as you expected; it would do what you wanted once the letter was discovered, except that it never will be, since if `letter.discovered` is `nil` what you get is not no remapping (in other words the execution of your `dobjFor(LookIn)` code), but the inherited remapping, which in this case is defined on `Room` as:

```
        dobjFor(LookIn) remapTo(Examine, self)
```

So if the letter is not discovered, **search study** or **examine study** will remap to **examine study** and your special case LookIn handling (to discover the letter) will never be invoked. What you actually need in this case is:

```
        dobjFor(LookIn)
        {
            remap = (letter.discovered ? [LookAction] : nil)
        }
```

Another situation when it's useful to use the underlying remap property directly is where you want different remappings depending on circumstances. For example you might want **push gate** to open the gate when it's closed, and close the gate when it's open. This is beyond the ability of `maybeRemapTo`, but quite possible with the underlying remap property:

```
gate: Door 'gate'  'gate'
  dobjFor(Push)
  {
     remap = (isOpen ? [CloseAction, self] : [OpenAction, self] );
  }
;
```

By the way, note that when we're writing the 'raw' code rather than using the macro, we have to give the full name of the Action class, hence `CloseAction` and `OpenAction` rather than just `Close` and `Open`. One of the things the `remapTo` and `maybeRemapTo` macros do for us is to add 'Action' to the name of the action we want (e.g. `Open` or `Close`), but if we're not using these macros, we have to do it ourselves.

A further complication with remap is that when it's used with two-object commands, special restrictions apply.

The first restriction is that a two-object command can only be remapped to another two-object command. Failure to observe this rule will result in a run-time error. The second restriction is that what we remap to must be an action applying to a specific object plus the placeholder `DirectObject` or `IndirectObject` (meaning the direct object or indirect object of the command we're remapping from). The specific object must be the one that's acting in the place of the object doing the remapping (which may be `self` if it's the same object, or else some other object), while the object role placeholder (`DirectObject` or `IndirectObject`) must reflect the role of the other object in the original command. Thus if `remapTo` follows `dobjFor()` it must use `IndirectObject` in its list of objects, and if it follows `iobjFor()` it must use `DirectObject`.

This should become clearer with an example. Suppose we have a desk with a single drawer. The drawer can be locked with a small brass key, but we want **lock desk with small brass key** to behave like **lock drawer with small brass key**. We could bring this about by defining the following on the desk:

```
dobjFor(LockWith) remapTo(LockWith, drawer, IndirectObject)
```

Similarly we might want **put something in desk** to be treated as **put something in drawer**, so we could write (on the desk object):

```
iobjFor(PutIn) remapTo(PutIn, DirectObject, drawer)
```

What we can't do is (for example) to decide that trying to put something in the desk is equivalent to dropping it and so write:

```
iobjFor(PutIn) remapTo(Drop, DirectObject) // WRONG !!!
```

This doesn't mean that you can't make trying to put something in the desk result in its being dropped on the floor, it just means you can't use remap to for that purpose (instead you'd have to write appropriate verify() and action() routines).

f. *Messages*

So far, when we've used things like `illogical()`, `reportFailure()` and the like, we've used them with single-quoted strings (e.g. `illogical('You can't do that. ')`). If you look in library source code, however, you won't see them used like that; instead you'll see them used with property pointers (e.g. `&cannotTakeMsg`) sometimes followed by one or more further arguments (a property pointer is & followed by the name of the property we want to reference). This allows the text of messages to be kept separate from the library code, which, among other things, makes it easier to translate the library into another language, since all the language-dependent stuff is on one place – or rather two (en_us.t and msg_neu.t) – rather than scattered all over the library.

What actually happens when the library sees something like `illogical(&cannotTakeMsg)` is that it calls the corresponding property on the object returned by `gActor.getActionMessageObj()`; thus, for example, when the library wants to display the message from `illogical(&cannotTakeMsg)` it actually calls:

```
gActor.getActionMessageObj().cannotTakeMsg
```

This then returns the single-quoted string to be displayed. This, incidentally, is why we need to use a property *pointer* here. A function call like `illogical(cannotTakeMsg)` would look for a property called `cannotTakeMsg` on the current object and pass the value of that property to whatever routine was going to process it. But that's not what we want here; what we want to do is to tell the routine which property of `gActor.getActionMessageObj()` to use, and for that we need to pass a property *pointer* (`&cannotTakeMsg`) not the property *value* (`cannotTakeMsg`).

When the player character is the actor (as is generally the case when executing player commands), then `gActor.getActionMessageObj()` returns `playerActionMessages`; if an NPC is performing the action then it returns `npcActionMessages`. Both objects are defined in msg_neu.t, where you can find all the library default messages.

Sometimes you'll see in library code that one of these message macros has more than one parameter; for example, you might see something like:

```
illogical(&mustBeHoldingMsg, self);
```

In such a case the second and subsequent parameters are arguments to the method invoked by the first parameter, so that the above example would get its message string from:

```
gActor.getActionMessageObj().mustBeHoldingMsg(self)
```

So far this may all seem quite remote from the concerns of a game author, but as we shall see, this mechanism can have its uses in game code.

Firstly, if you don't like any of the library default messages, you can simply modify the `playerActionMessages` object to substitute your own version (this is not the only way to do it, but it's probably as good as the alternative, which we'll see in a minute). For example, the standard response to trying to put something on an object that isn't a `Surface` is "There's no good surface on {the iobj/him}. "; thus, for example, if you try to put the stick on the cottage:

>put stick on cottage
There's no good surface on the pretty little cottage.

You might prefer a different message as a default, such as "You can't put anything on {the iobj/him}. " In which case all you need to do is to override the `notASurfaceMsg` in `playerActionMessages`:

```
modify playerActionMessages
    notASurfaceMsg = '{You/he} can\'t put anything on {the iobj/him}. '
;
```

If you like, you can try adding this to heidi.t, recompiling, and then trying putting the stick on the cottage (or anything else on anything else that's not a surface).

In addition to this, you can also change the messages by defining the appropriate message property on a class or object. Thus, for example, we could have obtained almost the same result by modifying Thing:

```
modify Thing
    notASurfaceMsg = '{You/he} can\'t put anything on {the iobj/him}. '
;
```

I say *almost* the same result because there is in fact a small (though readily fixable) catch with this that we'll come to shortly, which is why the first method might be slightly better for making global changes to messages. However, this second method is very useful when you want to customise the message that appears for individual objects (or particular classes of object). For example, suppose instead of the plain vanilla "You can't take that" message you'd get from trying to take the cottage, you'd like to see "It may be a small cottage, but it's still a lot bigger than you are; you can't walk around with it!" One way to do that would be to override the cottage's verify method for the Take action:

```
dobjFor(Take)
{
   verify()
   {
      illogical('It may be a small cottage, but it\'s still a lot
        bigger than you are; you can\'t walk around with it! ');
   }
}
```

That will work fine, but it's relatively verbose just for changing a message, and could quickly become quite tedious if you wanted to customize a lot of messages on a lot of objects. However, the mechanism we've just been exploring offers a handy short-cut; all you actually need to do to customize this response on the cottage is to add the following to its definition:

```
  cannotTakeMsg = 'It may be a small cottage, but it\'s still a
     lot bigger than you are; you can\'t walk around with it! '
```

What happens here is that the procedure for choosing which object will supply the message is a little more complex than originally described above. Before the parser selects either the `playerActionMessages` or `npcActionMessages` object it looks to see if the property it's looking for is defined on any of the objects defined in the action; if so, it uses that object instead. Since cottage now defines a `cannotTakeMsg` property, the cottage's version is used in preference to that defined on `playerActionMessages`.

But although this is very useful, it also presents a potential trap for the unwary. Suppose we also wanted to customise the response to **clean cottage**. In the same way we could just add a cannotCleanMsg property to the definition of the cottage:

```
+ Enterable -> (outsideCottage.in)
   'pretty little cottage/house/building' 'pretty little cottage'
    "It's just the sort of pretty little cottage that townspeople
    dream of living in, with roses round the door and a neat
    little window frame freshly painted in green. "

    cannotTakeMsg = 'It may be a small cottage, but it\'s
      still a lot bigger than you are; you can\'t walk around with it! '
    cannotCleanMsg = 'You don\'t have time for that right now. '
;
```

If you now recompile heidi.t and enter the command **clean cottage**, you'll see that it works as expected, you now see the response "You don't have time for that right now." The trouble is you'll see the same response if you enter the obviously nonsensical command **clean door with cottage**, instead of the expected "You can't clean anything with that" or "You wouldn't know how to clean that." The problem is that since the cottage defines a `cannotCleanMsg`, and the cottage is one of the objects involved in the command, the cottage's `cannotCleanMsg` is used in preference to any of the properties on playerActionMessages. The solution is to specify that you only want your custom `cannotCleanMsg` to be used when the cottage is the direct object of the command; you can do that with the `dobjMsg` macro:

```
        cannotCleanMsg = dobjMsg('You don\'t have time for that right now. ')
```

Similarly, if you defined a message that should only work when the cottage is the indirect object of a command (e.g. **put stick on cottage**), you must remember to use the `iobjMsg` macro:

```
        notASurfaceMsg = iobjMsg('You can\'t reach the roof. ')
```

Forgetting to use `dobjMsg` or `iobjMsg` when customizing a response for what is, or might be, a two-object command is a *very* easy mistake to fall into, so it's worth drumming into yourself at an early stage that you must *always* stop to think whether one or other macro might be necessary. Frequent use of customized messages is one thing that tends to distinguish a really good piece of IF from a mediocre one, so this is a technique you will want to master and use often in your own work.

There is one further problem here. If you type a nonsensical command like **clean cottage with door** you'll now see the response "You don't have time for that right now", which is clearly less than ideal. This can't be fixed simply by tweaking message properties; the cleanest solution here might be to make **clean with** fail in check rather than verify on the direct object, so the indirect object's failure message is used instead:

```
modify Thing
  dobjFor(CleanWith)
  {
    verify() {}
    check() { failCheck(&cannotCleanMsg); }
  }
;
```

The question you're probably asking yourself now is, "That's all very well, but how on earth do I know what message property I need to customise for a given action?" Well, you can find out by looking through the library code, but that's fairly laborious, so you really need a quick-reference chart: you should one find included in the TADS 3 documentation set, or a complete set of quick-reference charts for TADS 3 can be obtained from http://users.ox.ac.uk/~manc0049/TADSGuide/QRefs.zip. Since this information is so important for TADS 3 authors, it's also included in this *Guide* at Appendix A (though you'll probably find the downloaded chart a bit easier to use, since you can print it out on the two sides of a single sheet of paper). The chart doesn't list all the messages defined in the library (that would make such an unwieldy document that it would probably be self-defeating), but it does include the ones you'll most commonly want to override. For each of the transitive actions defined in the library (i.e. actions that take one or more Things as objects, but excluding actions such as Look or Inventory that don't), the chart shows the corresponding message property name as it is defined on Thing, and also on any subclasses where it is overridden to something different. The chart also indicates whether the message property is invoked from verify, check or action (abbreviated to V, C, and A respectively) and whether it is used when the object is either the direct or indirect object of the command (abbreviated to d or i). Thus, for example, if you look under PutIn (not to be confused with the former Russian president) you'll see:

PutIn	Thing iV	notAContainerMsg
	Fixture dV	cannotPutMsg
	Component dV	cannotPutComponentMsg(location)
	Immovable dC	cannotPutMsg

This means that `Thing.verifyIobjPutIn` uses `notAContainerMsg`, and this will propagate all the way down the class hierarchy (except for objects that *are* Containers, of course). There's no entry for Thing dV since in general there's no reason to rule out a Thing as the *direct* object of a PutIn command. However, since Fixtures, Components and Immovables can't be moved, they can't be put in anything, so there are messages for not being able to put them. The only difference between Fixture and Immovable is that in Fixture a PutInAction is ruled out in `verify()`, whereas in an

Immovable it's ruled out in `check()`; in both cases the `cannotPutMsg` is used. A Component also rules itself out as the direct object of a PutIn command, again in verify(), but this time with a different message and one that calls a parameter (location will normally be the object the Component is a component of). If you wanted to define your own `cannotPutComponentMsg` on an object, you can either simply define it as a single-quoted string, or as a method that's passed location as a parameter, e.g. either

```
cannotPutComponentMsg = 'You can\'t do that, because it's part of
  the worble-wangler. '
```

Or

```
cannotPutComponentMsg(obj)
{
    gMessageParams(obj);
    return 'You can\'t do that, because it\'s part of {the obj/him}. ';
}
```

As an example to try in the context of the Heidi game, you could try adding the following in the starting location (outsideCottage):

```
+ Distant 'forest' 'forest'
  "The forest is off to the east. "
  tooDistantMsg = 'It\'s too far away for that. '
;
```

This works fine, even though the library version of `tooDistantMsg` is actually a method which is passed self as a parameter (look under `Default` in Appendix A).

g. *Other Responses to Actions*

So far we have concentrated on how you can customise the responses to actions on objects directly involved in those actions as direct or indirect object, and that will probably be the most common type of action customisation you'll perform. But there are other types of response we should look at for the sake of completeness. (**NOTE:** This discussion assumes that `gameMain.beforeRunsBeforeCheck` is `nil`, otherwise stopping an action in `check()` will *not* prevent `beforeAction()` and `roomBeforeAction()` from being called on other objects, although everything else will be the same).

If you want an object not directly involved in a command to respond to it, you can use `beforeAction()` or `afterAction()`. As their names suggest, the first of these responds to the action just before it's performed (i.e. just before the appropriate action routine is invoked) while `afterAction()` is called just after an action is performed. These two routines are called on every object that's in the actor's scope when the action is performed, but only if the command reaches the action stage (i.e. it hasn't already been ruled out by `verify()`, `check()`, or `preCond`). Thus, if you want another object to respond to an action that fails, you must make it fail in `action()`, rather than before, e.g.

```
wickedWitch: Person 'wicked ugly witch' 'wicked witch'
  "Boy is she ugly! "
  isHer = true
  dobjFor(Kiss)
  {
     verify() { }
     action()
     {
         reportFailure('You move your lips towards hers, but your
           nerve fails you at the last moment. ');
     }
  }
;

bob: Person ' fine young man /bob'  'Bob'
   "He looks a fine young man. "
   isHim = true
   isProperName = true
   beforeAction()
   {
       if(gActionIs(Kiss) && gDobj == wickedWitch)
       {
           "<q>Hey, what do you  think you\'re doing!</q> cries Bob,
            grabbing you by the arm and pulling you back, <q>Don\'t you
           know that kissing her will turn you into a lump of vile green
           blancmange?</q> ";
           exit;
       }
   }
;
```

If the PC tries to kiss the wicked witch while Bob is present we'll get:

>kiss witch
"Hey, what do you think you're doing!" cries Bob, grabbing you by the arm and pulling you back, "Don't you know that kissing her will turn you into a lump of vile green blancmange?"

Whereas if the PC tries to kiss her when Bob is elsewhere we'll get:

>kiss witch
You move your lips towards hers, but your nerve fails you at the last moment.

Note the use of the `exit` macro in Bob's beforeAction() routine to veto the action before it takes place. We could alternatively have given Bob an afterAction() routine to give his reaction after the event:

```
  afterAction()
   {
       if(gActionIs(Kiss) && gDobj == wickedWitch)
       {
           "<q>Wise decision</q> Bob approves, <q>I suppose you realize
            that if you had gone ahead and kissed her you\'d have been
            turned into a lump of vile green blancmange!</q> ";
       }
   }
```

You can also allow the actor's location (Room or NestedRoom) to respond to actions in a similar way, but in that case you need to use roomBeforeAction() or roomAfterAction, e.g.:

```
topOfTree : OutdoorRoom 'At the top of the tree'
   "You cling precariously to the trunk, next to a firm, narrow branch."
    down = clearing
    enteringRoom(traveler)
    {
      if(!traveler.hasSeen(self) && traveler == gPlayerChar)
          addToScore(1, 'reaching the top of the tree. ');
    }
    roomBeforeAction()
    {
      if(gActionIs(Jump))
         failCheck('Not here -- you might fall to the ground and
            hurt yourself. ');
    }
    roomAfterAction()
    {
       if(gActionIs(Yell))
         "Your shout is lost on the breeze. ";
    }
;
```

Finally, you can also use the actorAction() method on the actor performing the action to interfere with or otherwise repond to an action the actor is about to perform. For example, suppose at some point in your game your player character is tied up, and while he's tied up he can't perform any actions other that system actions (like **quit**, **save** and **undo**) and **look**, **inventory** or **examine**; you might achieve this by adding an `isTiedUp` property to your Player Character object (normally `me`), and then adding the following `actorAction()` routine:

```
actorAction()
{
    if(isTiedUp && !gAction.ofKind(SystemAction)
        && !gActionIn(Look, Inventory, Examine)
     {
           "You can't do that while you're tied up. ";
            exit;
     }
}
```

Last of all, note that we've introduced a few more of those things beginning with g in the last page or so. Remember that `gAction` means the current action; `gActionIs(Whatever)` tests whether `gAction` is `Whatever` and returns true or nil accordingly; `gActionIn(Foo, Bar, Whatsit)` tests whether `gAction` is either `Foo` or `Bar` or `Whatsit` and returns true if it is or nil otherwise. The other one, `gMessageParams(obj)`, is a bit different; it's used to allow `obj` to be used in message parameter strings (like `'{the obj/him}'`) where `obj` is the name of an existing local variable. This allows us to write:

```
cannotPutComponentMsg(obj)
{
    gMessageParams(obj);
    return 'You can\'t do that, because it\'s part of {the obj/him}. ';
}
```

Instead of:

```
cannotPutComponentMsg(obj)
{
    return 'You can\'t do that, because it\'s part of '
       + obj.theName + '. ';
}
```

6. *Summary and Recapitulation*

A great deal of new material has been introduced in the course of this chapter, a lot of it in a not particularly systematic way as it has become necessary for implementing this or that game feature. It may be helpful to conclude the chapter with a slightly more systematic summary of the various library (and language) features employed.

a. *Connectors*

We have introduced various types of connectors. The simplest of these is simply the Room (or one of its subclassses). If a direction property of a Room is set to another Room, then traveling from the first room in that direction takes you to the second Room. The other types of connector we have used are:

```
FakeConnector
NoTravelMessage
OneWayRoomConnector
TravelMessage
```

These connectors have the following properties/methods that we have made use of:

```
destination
canTravelerPass (traveler)
explainTravelBarrier (traveler)
isConnectorApparent (origin, actor)
travelDesc
```

The `destination` property is relevant only to connectors that actually go somewhere (the `OneWayRoomConnector` and the `TravelMessage`); `travelDesc` applies only to `TravelMessage`.

Connectors can be assigned to direction properties as nested objects, e.g.:

```
myRoom : Room 'A Boring Room'
   "This room has a single exit through a narrow gap to the west. "
   west : TravelMessage
   {
       destination = myOtherRoom
       canTravelerPass(traveler) { return !bigHeavyPlank.isIn(traveler); }
        explainTravelBarrier(traveler) { "You can't get through
            the narrow gap carrying that great big plank. "; }
        travelDesc = "You just manage to squeeze through the gap. "}
   }
   north : NoTravelMessage { "Try as you might, you can't walk through
      the wall. "}
   south : FakeConnector { "On second thoughts you decide against going
    through the fourth-floor window. "}
;
```

For further information on connectors, have a look at the *TADS 2 Tour Guide* and the *TADS 3 Library Reference Manual*.

b. *Rooms*

In addition to the basic Room properties introduced in the previous chapter and the direction properties used in this chapter, we have made use of the following methods of Room objects /macros:

```
enteringRoom(traveler)
asExit()
```

c. *Object Types & Properties*

The various types (or, more accurately, classes) of game object we have made use of in this chapter include:

```
Thing
Surface
Container
Fixture
Enterable
FloorlessRoom
Hidden
PresentLater (mix-in class)
Chair
```

(The last of these is a subclass of `NestedRoom`.)

We made use of (among others) the following object methods/properties:

- `initSpecialDesc` (description of item not yet moved)
- `moved` (has the object moved from its initial location?)
- `lexicalParent` (the object in which a nested object is nested)
- `isIn()` (to determine if one object is contained within another)[27]

d. *Dealing with Actions*

We have seen how the `dobjFor` and `iobjFor` macros, in conjunction with `preCond`, `verify()`, `check()`, `remap` and `action()`, may be used to define the behaviour of game objects that are the direct or indirect objects of particular commands. For further details see the *Technical Manual*.

We have also seen how dobjFor may be used in conjunction with other macros to redirect or redefine actions, e.g.

```
dobjFor(Search) asDobjFor(Examine)
dobjFor(Climb) remapTo(TravelVia, clearing.up)
```

Also the following may be used to stop processing the current action and carry out another instead, or to carry out one action within another:

```
replaceAction(Take, stick)
nestedAction(Take, stick)
```

[27] Note that this returns true even if the containment is indirect. For example if the pencil is the in drawer, and the drawer is the desk, and the desk is in the study, then pencil.isIn(study) is true. To test for *direct* containment use isDirectlyIn().

To test for what type of action (command) is being currently executed use, e.g.:

```
if(gActionIs(Drop))
```

To block an action at the verify stage use:
`illogical ('Reason why this cannot be done')`
or
`illogicalNow('Reason why this cannot be done right now')`

To block it at the `check()` or `action()` stage, use `exit` or `failCheck()`.

To have other objects, the location, or the actor interfere with or respond to the action, use `beforeAction()`, `afterAction()`, `roomBeforeAction()`, `roomAfterAction()`, or `actorAction()`.

Override message properties (e.g. `cannotTakeMsg`) to customise responses.

e. *Miscellaneous*

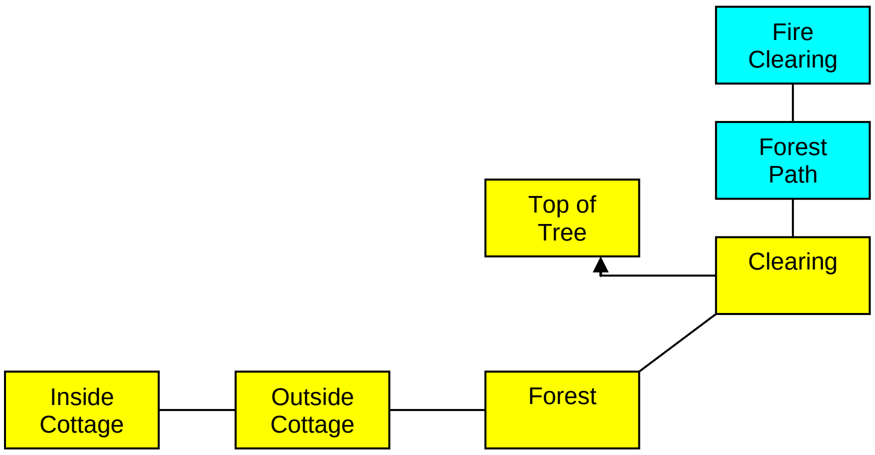
Other things we made use of include:
- `addToScore(points, 'what you have done to deserve them')`
- `gPlayerChar` (the Player character object)
- `gActor` (the actor carrying out the current command)
- `gDobj` (the direct object of the current command)
- `gIobj` (the indirect object of the current command)
- `inherited` (to invoke the behaviour defined on a superclass)
- `{The dobj/he}` (an example of message parameter substitution)

Chapter Five - Character Building

1. *Setting the Scene*

The main task in this chapter will be to add an NPC (Non-Player Character) to our game, though in the course of doing so we shall be looking at a number of other matters. The first task is to add another couple of locations to give our NPC somewhere to operate. He's going to be a charcoal-burner working in the forest; clearly, then, he needs a fire to tend, which we'd better put in another clearing. In order to avoid having one clearing running straight into another, we'll put a length of path in between.

Once again, you might like to have a go at implementing all this yourself before turning the page and seeing how this guide does it. First of all you need to add a 'forest path' room north of the clearing, and a 'fire clearing' room north of the forest path, remembering to add all the appropriate connections. The game map should then look like this, with the new rooms you're adding shown in pale blue:



There's no need to put any objects in the Forest Path, but in the Fire Clearing we'll want a fire, and also the smoke given off by the fire. Give some thought to what class to make these objects. In particular, smoke is not a solid physical object, so you might want to make it of a class we haven't encountered before, `Vaporous`, since this has the kind of behaviour we need; it is designed for insubstantial objects such as fire, smoke and fog which you can sense but not interact with in any other way.

It is not quite right for the fire in the clearing, however, since this fire is something rather more substantial. The fire object will require some thought since quite apart from the fact that it's too big to pick up or push around, there are more immediate reasons why one would not expect Heidi to try to manipulate a burning fire. It would be tedious for you to have to write special handling for every single action the player might on the fire, however, so it may help you to know that there's a short-cut way of dealing with this: you can use `dobjFor(Default)` (which means, this is what we do when the current object is the direct object of any action not explicitly defined for this object). There are some actions you will then need to explicitly allow, however, such as examining and smelling the fire.

Our new code looks like this:

```
forestPath : OutdoorRoom 'forest Path'
  "This broad path leads more or less straight north-south
   through the forest. To the north the occasional puff of
   smoke floats up above the trees. "
   south = clearing
   north = fireClearing
;

fireClearing : OutdoorRoom 'Clearing with Fire'
  "The main feature of this large clearing a large, smouldering charcoal
   fire that periodically lets off clouds of smoke. A path leads off
   to the south, and another to the northwest. "
   south = forestPath
   northwest : FakeConnector {"You decide against going that way
     right now. "}
;

+ fire : Fixture 'large smoking charcoal fire' 'fire'
   "The fire is burning slowly, turning wood into charcoal. It nevertheless
   feels quite hot even from a distance, and every now and again lets out
   billows of smoke that get blown in your direction. "
   dobjFor(Examine)
   {
     verify() {}
     action() { inherited; }
   }
   dobjFor(Smell) remapTo(Smell, smoke)
   dobjFor(Default)
   {
     verify() { illogical('The fire is best left well alone; it\'s
     <i>very</i>  hot and {you/he} do{es}n\'t want to get too close. ');}
   }

;

+ smoke : Vaporous 'smoke' 'smoke'
  "The thick, grey smoke rises steadily from the fire, but gusts of wind
    occasionally send billows of it in your direction. "
  smellDesc = "The smoke from the fire smells acrid and makes you cough. "
;
```

There's one further change we need to make before trying any of this out, and that's to change the `north` property of `clearing` to read:

```
north = forestPath
```

There's not a lot here that's new in principle, but one or two things about the fire and smoke objects merit some further explanation.

So we start by defining the fire as a `Fixture`, since it certainly isn't the sort of thing one would walk away with. We have used `dobjFor(Default)` to stop most actions on the fire at the verify stage since this makes reasonably good conceptual sense, it should seem illogical to the player to take, eat, or move a fire; it also conveniently stops it before the display of default messages defined in the `check`() or `action`() methods of `dobjFor` the various verbs on any of Fixture's superclasses, such as `Thing`. But, since left to itself, `dobjFor(Default)` would trap *all* actions on the fire, we need to make Examine carry out its inherited behaviour. Finally, since it would not be unreasonable to smell the fire, we allow for that also, in this case by redirecting the action to the smoke emanating from the fire.

2. *A Basic Burner*

Now that we have set the scene, we can introduce our NPC, a charcoal burner who will be tending the fire. We may start by defining him thus:

```
burner : Person 'charcoal burner' 'charcoal burner'
  @fireClearing
  "It's rather difficult to make out his features under all the grime and
    soot. "
  properName = 'Joe Black'
  globalParamName = 'burner'
  isHim = true
;
```

This may seem very simple code for such a potentially complicated object, and it certainly doesn't look like the charcoal burner will do very much. The main reason for the simplicity of this object definition is that most of the complexity of NPCs will be handled through `ActorState` and `TopicEntry` objects, which we'll be encountering shortly. Strictly speaking, the TADS 3 library doesn't *force* you to use ActorStates and TopicEntries; you are free if you wish to code your NPC with `dobjFor(This)` and `iobjFor(That)` and a host of `switch` and `if` statements, but unless your NPC is fairly simple, this is likely to result in tangled spaghetti code that becomes harder and harder to maintain. Since we're not trying to create Burner Bolognese we'll stick to the means provided by the library, which allows highly sophisticated NPC behaviour with code that's both much cleaner and easier to understand and maintain. The secret is that we go about coding our NPC using a largely *declarative* rather than a largely *procedural* approach; in other words we define the NPC's behaviour through a series of object definitions rather than through a mass of code controlled by state variables, switch statements and the like.

But before moving on to see how this declarative approach works, let's stop and look at our basic burner object. The first thing to note is that he's of class `Person`, which seems pretty reasonable. `Person` is (indirectly) a subclass of the more generic `Actor` class; we use `Person` rather that `Actor` since an `Actor` could be a small furry animal you could pick up and carry around with you, which our charcoal burner certainly isn't.

Secondly, we have defined the initial location of our burner object using `@fireClearing` rather than the + syntax (which would have worked perfectly well). There are two reasons for doing it this way: (a) ActorStates and TopicEntries are all objects that will be located in their Actor (either directly, or more deeply nested to several levels); had we put a + before the definition of our actor that would be one more + we'd have to put before each and every `ActorState` and `TopicEntry` object – not that much more typing, perhaps, but something that would make our code that much less readable and more error-prone, especially if we end up having to nest to ++++ or +++++; (b) in a more complex game we might want to move our NPC definition (together with all its associated objects) to a different place in our code, or even into a different source file; if we had defined the NPC's starting location with the + syntax we'd then not only have to remove the + from in front of the NPC, but remove one + from each and every `ActorState` and `TopicEntry` we'd nested inside it.

Thirdly, the `properName` property is not part of the TADS 3 library at all; it's a property we've defined on the object for our own use. At the moment this simply illustrates that this is something we can do. What this new property is for hardly needs explaining; how we are going to use it is something we shall reveal shortly.

Fourthly, the `globalParamName` is just a convenience feature that we shall use shortly. What it allows us to do is to use parameter substitution strings like `{the burner/he}` when we want to display the current name of the burner (which will change from "the charcoal burner" to "Joe Black" once he's introduced himself). Finally, defining `isHim = true` means that the charcoal burner can be referred to as 'him'. For a female NPC you'd define `isHer = true`.

Both `TopicEntry` and `ActorState` are generic classes with several subclasses that tend to be the classes one uses in practice. At some point you should look at the more detailed explanation of their use in the "Creating Dynamic Characters" articles in the *Technical Manual*, but for now you can just follow the discussion here.

Since the main objective of the game is to return the diamond ring to the charcoal burner (for reasons that will become apparent later), we may as well start by making our burner respond when Heidi gives him the ring. In the bad old days we should have had to do that by writing `dobjFor(Give)` methods on the ring and `iobjFor(Give)` methods on the burner (and likewise for Show, if we wanted the burner to respond to being shown the ring). Fortunately, this can now all be handled by TopicEntries, more specifically, by the subclasses of `TopicEntry` called `GiveTopic`, `ShowTopic` and `GiveShowTopic`. As their names suggests, a `GiveTopic` defines its actor's response to a Give command, a `ShowTopic` to a Show command, and a `GiveShowTopic` to either a Give or a Show command. For our purposes, we may as well use a `GiveShowTopic`.

```
+ GiveShowTopic @ring
  "As you hand the ring over to {the burner/him}, his eyes light up in
    delight and his jaws drop in amazement. <q>You found it!</q> he
    declares, <q>God bless you, you really found it! Now I can go and call
    on my sweetheart after all! Thank you, my dear, that's absolutely
    wonderful!</q>"
;
```

Once again, using a template makes defining this object very simple. The object following the @ symbol is not the location but in fact the value of the `GiveShowTopic's matchObj` property, which means the object that is the direct object of the Give or Show command matched by this TopicEntry. In other words, this `GiveShowTopic` will be invoked in response to the commands **give ring to burner** or **show ring to burner**. The double quoted string is the value of the `GiveShowTopic's topicResponse` property, which is displayed when the `GiveShowTopic` is invoked. We precede the object definition with a + sign because a `TopicEntry` has to be contained within its Actor (or one of its Actor's ActorStates or TopicGroups). The `<q>` and `</q>` sequences within the `topicResponse` string are codes for opening and closing smart quotes: when the game is run they should be displayed thus: "You found it!" he declares, "God bless you…"[28] Likewise `{the burner/him}` is a parameter string that should display as either 'the charcoal burner' or 'Joe Black' depending on whether we know his name at that point (how we learn his name is something we'll be seeing later).

That will work if Heidi hands Joe his ring, but what happens if she tries to give or show him anything else? For now, the only thing he's interested in receiving from Heidi is his ring, so if she tries to give him anything else, he should refuse. But if he always refuses in precisely the same way he'll begin to look a bit wooden. The

[28] Unfortunately you'll just have to imagine the curly quotes here, as my PDF converter seems to be unable to cope with them.

best way to handle this, then, is through a combination of a `DefaultGiveShowTopic` (which defines the response to Joe's being shown or given anything not otherwise specifically defined) and a `ShuffledEventList`, which picks a random response from a list (more on which shortly).

```
+ DefaultGiveShowTopic, ShuffledEventList
  [
    '{The burner/he} shakes his head. <q>No thanks, love.</q>',
    'He looks at it and grins. <q>That\'s nice, my dear,</q> he remarks,
     handing it back. ',
    '<q>I\'d hang on to that if I were you,</q> he advises. '
  ]
;
```

This definition means that the object is both a `DefaultGiveShowTopic` and a `ShuffledEventList`. The list of single-quoted strings between the square brackets is the `eventList` property of the `ShuffledEventList` (via the `DefaultTopic` template). Notice that these strings must be separated by commas, and they must be single-quoted, not double-quoted, strings. This, incidentally, is another reason why the `{The burner/he}` syntax is valuable, it can be used in both types of string, whereas the alternative way of achieving the same effect, `<<burner.theName>>`, can only be used in double-quoted strings. Since the strings are single-quoted we have to use the backslash character (\) before any apostrophes we want to use inside them (hence "That\'s nice" for "That's nice").

The `ShuffledEventList` uses the strings from its list in random order, but does not repeat any one string until it has used all of them; it is thus like shuffling a pack of cards, turning each card over in turn, then repeating the process (as often as desired). This is better for the purpose than a `RandomEventList` which could in principle print the same string two (or more) times in succession.

You can now try recompiling the game and playing it through to the point where you hand the ring to the burner (you can try handing him other objects first to see what happens). Everything should work fine apart from one thing – returning the ring to the charcoal burner is meant to be the object of the game, but apart from the burner's effusive thanks, nothing much happens when the ring is handed over. The game just carries on and the player isn't even awarded any extra points. In order to fix this, we'll take a brief detour through a special function for ending games.

3. *Ending the Game*

TADS 3 provides a `finishGameMsg(msg, extra)` function for ending a game and displaying a message. This function optionally displays a message explaining precisely how or why the game has ended, such as *** YOU HAVE DIED *** or *** YOU HAVE WON ***. Although there will not be many ways of ending the game in *The Further Adventures of Heidi*, and we won't let Heidi get killed off, we can demonstrate the use of this funtion at one point in the game: when the player wins.

What the function does is to end the game, display a message explaining why, and then provide the user with options to RESTORE, RESTART, or QUIT, plus any additional options such as UNDO or show the FULL SCORE defined by the `extra` parameter (which is passed as a list). So, for example, if you want the UNDO option and the FULL SCORE option to be displayed, specify the second argument to `finishGameMsg` as `[finishOptionUndo,finishOptionFullScore]`.

The first argument, `msg`, can either be a single quoted string containing the message you want displayed, such as 'YOU HAVE FAILED DISMALLY IN YOUR QUEST' or one of the pre-defined `FinishType` objects: `ftDeath`, `ftVictory`, `ftFailure` or `ftGameOver`, which display an appropriate message (you could also define your own `FinishType` objects, but that's a complication we'll leave for now). Either way the message will appear surrounded by asterisks ('***'). Alternatively, if the msg argument is nil no message will be displayed.

If you were going to call this function from several different places in your code, always with the same options, you might find it convenient to define your own wrapper function to do this, for example:

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

Then at each point where you wanted the game to end, you could simply call `endGame(msg)` without having to specify the list of extra options that you always wanted to be displayed. Since, however, `finishGameMsg` is only called once in heidi.t, there's little point our doing that here.

That's all very well, but we now need to call `finshGameMsg()` when Heidi hands the ring over to the charcoal burner, and at first sight that looks a bit tricky because it appears that all the `GiveShowTopic` does is to display a string.

One way to get round this would be to place `<<finshGame(ftVictory, [finishOptionUndo,finishOptionFullScore]))>>` at the end of the double-quoted string in the `GiveShowTopic`. Since a function call is a perfectly valid expression, and this one effectively returns nil, this should work perfectly well. However, we might also want to add a couple of points to the player's score at this point, at which point using the <<>> construct starts to get a bit cumbersome. Besides, it's useful to have another approach up our sleeve for situations where embedding expressions in double-angle brackets won't really do the job.

What we do is simply to exploit the fact that although the library expects `topicResponse` to be a property containing a double-quoted string, the TADS 3 compiler will be perfectly happy if we treat it as a method containing any code we like.[29] We can thus amend our definition of the `GiveShowTopic` to:

```
+ GiveShowTopic @ring
  topicResponse
  {
    "As you hand the ring over to {the burner/him}, his eyes light up in
      delight and his jaws drop in amazement. <q>You found it!</q> he
      declares, <q>God bless you, you really found it! Now I can go and call
      on my sweetheart after all! Thank you, my dear, that's absolutely
      wonderful!</q>";
     addToScore (2, 'giving {the burner/him} his ring back. ');
     finishGameMsg(ftVictory, [finishOptionUndo,finishOptionFullScore]);
  }
;
```

[29] You could instead override the method handleTopic(fromActor, topic), which is the method that in turn invokes the topicResponse property (or method), but since this requires you to remember and type an argument list you probably won't use, there seems to be little or no advantage to it.

We have finally reached the point where the game is playable all the way through. It's not a very exciting game, to be sure, but at least it's now winnable. It would be more interesting if we could make the charcoal burner a more responsive character, so the player could learn a little more about him, how he came to lose the ring, why it's so important to him, and so forth. That is what we shall try to do next.

4. *The Art of Conversation*

When you go up and talk to someone, the chances are that they won't just carry on what they're doing while they're talking with you, they're more likely to stop and adopt another posture (no doubt you can think of plenty of exceptions to this, but it's true most of the time). Also, it's normal to begin and end a conversation with some kind of greeting and farewell protocol (e.g. saying "Hello and goodbye."). Of course there are people who will just bounce up to you and say, "Have you done the monthly sales figures yet?" or "What do you think about the election results?", but it's more normal to start with "Good morning" or the like.

The traditional way of programming NPCs to respond to ASK ABOUT and TELL ABOUT (as in **ask jones about monthly report** or **tell fred about election results**) doesn't really allow for such niceties. The player character just walks up to the NPC and tries to find out what topics he, she, or it will respond to, without much sense that a conversation is starting or ending in the normal way. TADS 3 goes a long way to providing a more realistic approximation to the way human beings actually converse by using ActorStates and greeting protocols.

The idea is that an actor (NPC) typically starts in a type of ActorState called a `ConversationReadyState` that defines what he or she is doing prior to the conversation, and how the conversation is begun and ended. Starting a conversation with the actor (either with a **talk to** command, or by using **ask about**, **ask for**, **tell about**, **give to**, or **show to**) then causes the greeting message to be displayed, and the actor to switch into the corresponding `InConversationState`. This latter type of `ActorState` typically provides a description of what the actor is doing while talking to you, and contains within it the various `TopicEntry` objects to which the actor will respond while in that state.

To see how this works in practice, add the following code immediately after the definition of the `DefaultGiveShowTopic`:

```
+ burnerTalking : InConversationState
  stateDesc = "He's standing talking with you. "
  specialDesc = "{The burner/he} is leaning on his spade
    talking with you. "
;

++ burnerWorking : ConversationReadyState
  stateDesc = "He's busily tending the fire. "
  specialDesc = "<<a++ ? '{The burner/he}' : '{A burner/he}'>>
  is walking round the fire, occasionally shovelling dirt onto it with his
    spade. "
  isInitState = true
  a = 0
;

+++ HelloTopic, StopEventList
  [
    '<q>Er, excuse me,</q> you say, trying to get {the\'s burner/her}
      attention.<.p>
```

```
     {The burner/he} moves away from the fire and leans on his spade
     to talk to you. <q>Hello, young lady. Mind you don\'t get too
     close to that fire now.</q>',
    '<q>Hello!</q> you call cheerfully.<.p>
     <q>Hello again!</q> {the burner/he} declares, pausing from
     his labours to rest on his spade. '
  ]
;

+++ ByeTopic
  "<q>Bye for now, then.</q> you say.<.p>
   <q>Take care, now.</q> {the burner/he} admonishes you as he
     returns to his work. "
;

+++ ImpByeTopic
  "{The burner/he} gives a little shake of the head and returns
    to work. "
;
```

The `specialDesc` is the description of the actor that appears in the room description. The `stateDesc` is appended to the end of the `desc` of the actor when he's examined with an **examine** command. By default, the library expects to find the `ConversationReadyState` contained within its corresponding `InConversationState` (which is what we have done here). Clearly, the game needs to know which `ActorState` our charcoal burner starts in; we achieve that by setting `isInitState = true` on `burnerWorking` (the `ConversationReadyState`).

To display what happens at the start of a conversation, we use a `HelloTopic` located in the `ConversationReadyState`. In this game, we are assuming that Heidi and the charcoal burner have never seen each other before, so the exchange between them on their first meeting is likely to be different from that on subsequent occasions. We accordingly define the `HelloTopic` to be a `StopEventList` as well. A `StopEventList` works through each element in its list in sequence, until it reaches the last one, which it then keeps repeating. In this example we have only provided two strings in the list – one for the first greeting and a second one for every subsequent greeting. We could also use an `ImpHelloTopic` to provide a different response if Heidi strikes up a conversation with Joe without first explicitly greeting him through a player command (**talk to burner**), but this is a complication we can manage without here.

Likewise, to display what happens at the end of the conversation we use a `ByeTopic`. We could once again display a list of different messages when the conversation is terminated, but here we take the simpler option of displaying the same message each time. On the other hand we supply a separate `ImpByeTopic` to define what happens if the conversation is ended implicitly (either by Heidi walking away in the middle of the conversation, or by exhausting the burner's attention span by failing to continue the conversation) rather than explicitly (with a **bye** command).[30]

If this all seems a bit much to take in, it may become clearer if you try running the game again with it included and seeing how the charcoal burner now behaves (you can get into conversation with him either explicitly with **talk to burner** or by trying to give him or show him something). At some point you will also want to have a careful read of the articles on creating dynamic characters in TADS 3 in the *Technical Manual*, though there's no need to do that until you've completed this guide.

[30] You could further distinguish between these two implicit goodbye cases with LeaveByeTopic and BoredByeTopic. See the *TADS 3 Tour Guide* for details.

I should explain, though, that there's one little trick in the code given above that you won't find documented there or anywhere else. Since Heidi has never seen the charcoal burner before, the very first time he's mentioned he should be described as "a charcoal burner", but, once he's been referred to once, on every subsequent occasion he should be called "the charcoal burner" (until we learn his name, when he'll be referred to as "Joe Black", which is why we're using the substitution strings – `{the burner/he}` and so forth). To achieve this effect, the `specialDesc` property of burnerWorking has been defined thus:

```
specialDesc = "<<a++ ? '{The burner/he}' : '{A burner/he}'>>
  is walking round the fire, occasionally shovelling dirt onto
   it with his spade. "
```

The first part of this little trick is the use of the ternary operator `? :` . This means if the expression before the question-mark is true, evaluate to the expression between the question-mark and the colon, otherwise evaluate to the expression after the colon. So, for example:

```
       (x > 5) ? 96 : 32
```

evaluates to 96 if x is 6 but to 32 if x is 4. Moreover, `a++` means use the current value of `a`, then increase it by one after we've use it. We have initialized the property a to 0 (something as uninformative as 'a' would normally be frowned upon as a property name, but since we're using it here to determine whether the burner should be called *a* burner or *the* burner, we might just about be forgiven for it). This means that the first time the `specialDesc` string is displayed, `a` is zero (which is treated as equivalent to `nil`, i.e. a Boolean false), so the whole expression in the double angle brackets evaluates to the parameter string `'{A burner/he}'` which in turn displays as "A charcoal burner". But since the act of displaying the string causes `a` to increment by one, on every subsequent occasion the `specialDesc` string is displayed `a` will be some number greater than zero, which will be treated as meaning `true`. This will result in the whole expression evaluating to `'{The burner/he}'`, and thus displaying as "The charcoal burner".

The next step is to give the charcoal burner a couple of topics he can talk about. Since he's tending a fire, and the fire and smoke are mentioned in the room description, they might be two obvious topics to start with. Compared with what we've just done, coding them is fairly straightforward:

```
++ AskTopic @smoke
   "<q>Doesn't that smoke bother you?</q> you ask.<.p>
   <q>Nah! You get used to it - you just learn not to breathe too deep when
    it gets blown your way.</q> he assures you. "
;

++ AskTopic, ShuffledEventList @fire
   [
     '<q>Why have you got that great big bonfire in the middle of the
       forest?</q> you ask.<.p>
     <q>It\'s not a bonfire, Miss, it\'s a fire for making charcoal.</q> he
      explains, <q>And to make charcoal I need to burn wood - slow like -
      and a forest is a good place to find wood - see?</q>',
     '<q>Doesn\'t it get a bit hot, working with that fire all day?</q> you
        wonder.<.p>
     <q>Yes, but it beats being cooped up in an office all day.</q> he
       replies, <q>I couldn\'t stand that!</q>',
     '<q>Why do you keep putting that dust on the fire?</q> you wonder.<.p>
     <q>To stop it burning too quick.</q> he tells you. '
   ]
;
```

We make them both of class `AskTopic` so that the burner will respond to **ask burner about fire** or **ask burner about smoke**. For the smoke we've just given him a single reply he'll give every time (not because this is a particularly good idea, but just to show how it's done). For the fire, we've given him a list of three responses which will be treated as a `ShuffledEventList` (since their order is not significant). There's only a couple of other points to note here. The first is that our response strings define both sides of the conversation, so we see what Heidi asks as well as what the burner answers. The second is that these topics belong in `burnerTalking`, the `InConversationState`, so we precede them both with two plus signs to contain them at the right place in the object hierarchy.

It's always possible that the player will try to ask the burner some topic we haven't explicitly defined, so it would be useful to define a catchall `DefaultAskTellTopic` to handle such cases:

```
++ DefaultAskTellTopic
   "<q>What do you think about <<gTopicText>>?</q> you ask.<.p>
   <q>Ah, yes indeed, <<gTopicText>>,</q> he nods sagely,
   <q><<rand('Quite so', 'You never know', 'Or there again, no
    indeed')>>.</q>"
;
```

This definition (loosely based on a similar trick in the sample game that comes with TADS 3), is designed to create the vague illusion of responding to any topic (though the illusion will quickly be shattered in practice), by using `gTopicText`, which returns the text of whatever the player typed after **about** in an **ask burner about** or **tell burner about** command. When the `rand()` function is given a list of arguments, as here, it selects one of them at random; this at least gives a measure of variety to the charcoal burner's meaningless replies, and will generate a transcript like:

>**ask burner about the weather**
"What do you think about the weather?" you ask.

"Ah, yes indeed, the weather," he nods sagely, "You never know."

>**ask him about weapons of mass destruction**
"What do you think about weapons of mass destruction?" you ask.

"Ah, yes indeed, weapons of mass destruction," he nods sagely, "Or there again, no indeed."

>**ask him about his mother**
"What do you think about his mother?" you ask.

"Ah, yes indeed, his mother," he nods sagely, "Or there again, no indeed."

The last of these rather gives the game away, which is why this particular technique (trying to echo what the player typed in the player character's response) is not really satisfactory, unless you're trying to represent your NPC as an eccentric old buffer who displays his confusion by talking like this. It's usually rather better to show non-commital responses to the effect that the NPC doesn't hear the question, refuses to answer, mutters an inaudible reply, becomes distracted, or says something so convoluted that you fail to understand it. In such cases it isn't really practicable to give the full question asked by the player character, so you either have to omit it, or represent it in indirect speech (e.g. "You ask your question and…") or have the NPC interrupt it half way through, (e.g. "What do you think about…?" you begin. "I don't," he interrupts, "Thinking's for intellectuals, and I sure as hell ain't one of them.") As an exercise you may wish to devise your own list of default ask responses for our burner to replace the slightly dodgy ones shown above.

Now the time has come to get the charcoal burner to tell us about himself (in response to the player typing **ask burner about himself**). This doesn't require us to define a special himself object or topic, since the parser will recognize **ask burner about himself** as equivalent to **ask burner about burner**. We simply need to add an `AskTopic` with `burner` as its `matchObj`:

```
++ AskTopic @burner
   "<q>My name's Heidi.</q> you announce. <q>What's yours?</q><.p>
   <q><<burner.properName>>,</q> he replies, <q>Mind you, it'll soon be
    mud.</q>"
;
```

For clarity of code structure, it might be a good idea to put this just before our catchall `DefaultAskTellTopic`. The one point to note here is the use of `<<burner.properName>>` rather than simply Joe Black. The advantage of doing it this way is that if we later decide we want to call the charcoal burner 'Fred Bloggs' or 'Ebenezer Oddball Sidewinder Bumblebotham' instead, we need only change the value of the `burner` object's `properName` property, rather than having to hunt down and change every occurrence of the string 'Joe Black'.

Joe's statement (let's stick with calling him Joe) that his name will soon be mud invites the question why, but this doesn't seem to be the sort of question that naturally fits the **ask about** format: **ask joe about mud**, for example, wouldn't read right. What we'd really like is to be able to **ask why**, but for this to be a valid question only at this point in the conversation. Fortunately TADS 3 makes this possible through a mechanism called *conversation nodes* used in conjunction with `SpecialTopics`. A conversation node represents a particular point in the conversation (such as here, when Joe tells us his name will be mud) at which particular responses make sense which might not make sense elsewhere. Another example might be when an NPC asks a question requiring a yes or no answer: it makes sense to answer yes or no at that point, but it probably would have made no sense to do so on the previous turn, and the moment when it makes sense may have passed by the next turn. For something more complicated than a straightforward 'yes' or 'no' response, we can define a `SpecialTopic`, which in principle allows the player character to ask any question or make any reply we like (though in practice we'll want to restrict their complexity), with the restriction that these responses are only valid while their Conversation Node is active (because after that the conversation will have moved on, and before that the NPC hadn't yet made the remark to which these are potentially relevant responses).

This may become clearer with an example. What we want to do is to allow Heidi to ask why Joe thinks his name will be mud. To do this we need to define a conversation node (which we'll call 'burner-mud') and add a couple of SpecialTopics to it to handle questions like **ask why**. We also need to tell the game when to enter the 'burner-mud' conversation mode. This can be done by use of the `<.convnode name>` tag, where `name` is the name of the conversation node we want to enter. We use it by including it in the output string at the appropriate point:

```
++ AskTopic @burner
   "<q>My name's Heidi.</q> you announce. <q>What's yours?</q><.p>
   <q><<burner.properName>>,</q> he replies, <q>Mind you, it'll soon be
   mud.</q> <.convnode burner-mud>"
;
```

The definition of the `ConvNode` object is pretty minimal. The SpecialTopics require a little more attention, and we add a `DefaultAskTellTopic` at the end to ensure that the player stays in the `ConvNode` until he or she asks the question we want asked:

```
+ ConvNode 'burner-mud';

++ SpecialTopic, StopEventList
   'deny that mud is a name'
   ['deny', 'that', 'mud', 'is', 'a', 'name']
   [
     '<q>Mud! What kind of name is that?<q> you ask.<.p>
     <q>My name -- tonight.</q> he replied gloomily. <.convstay>',
     '<q>But you can\'t <i>really</i> be called <q>Mud</q></q> you
       insist.<.p>
     <q>Oh yes I can!</q> he assures you. <.convstay>'
   ]
;

++ SpecialTopic 'ask why' ['ask', 'him', 'why']
  "<q>Why will your name be mud?</q> you want to know.<.p>
  He shakes his head, lets out an enormous sigh, and replies,
  <q>I was going to give her the ring tonight -- her engagement ring --
  only I've gone and lost it. Cost me two months' wages it did. And
  now she'll never speak to me again,</q> he concludes, with another
  mournful shake of the head, <q>never.</q>"
;

++ DefaultAskTellTopic
  "<q>And why does...</q> you begin.<.p>
  <q>Mud.</q> he repeats with a despairing sigh. <.convstay>"
;
```

Note that there is only one + sign in front of the `ConvNode`. This is because here we are putting the `ConvNode` directly inside the actor (in this case the burner), not any of its actor states. It is also legal (though never necessary) to locate a `ConvNode` in an `ActorState`.

We should spend a few minutes thinking about how all this works. First, since the player cannot be expected to guess what wording will trigger our special topics, we need to provide some kind of prompt. This is what the single-quoted strings immediately after the class names do (the strings in the `name` property of the `SpecialTopic` objects). These strings need to be of a form that make sense after "You could…"; in this case the player will be prompted with:

(You could deny that mud is a name, or ask why.)

The list of strings in square brackets (the `keywordList` property) is then the list of words (or 'tokens') that the parser will check for in deciding whether to match a given `SpecialTopic`. It is not necessary that the player types all the words in the list for a match to take place – in this instance one or other of the SpecialTopics will be matched if the player simply types **deny** or **mud** or **why**, for example; but a match will not take place if the command contains any words not in the list. For example if the player types **deny mud a proper name** the parser will respond with:

The word "proper" is not necessary in this story.

The only way to try to avoid this it to be as careful as possible in the list of strings you include in the `keywordList`; in this instance we have mostly just duplicated the words that will appear in the prompt, but it seems likely that the player might try something else (and you can be sure than many players will), we can always add more words to these lists (e.g. 'him' between 'ask' and 'why'). Note also that the list of the words in the `keywordList` property must be separated by commas, which can often be surprisingly easy to forget.

Everything else about the SpecialTopics works in the same way as for other TopicEntries. We supply the 'deny mud is a name' topic merely to give the player the appearance of having an option at this point; a prompt that merely said "(You could ask why)" would look a bit *too* directive. We use the `<.convstay>` tags in the 'deny mud' topic and the `DefaultAskTellTopic` to try to prevent the player from leaving the conversation node until he or she has asked why and so given Joe a chance to start telling his sorry tale (the player could just walk away and terminate the conversation without learning about the ring, but one hopes that most players' natural curiosity will prompt them to **ask why** first).

Since the player could type **deny mud is a name** more than once, we provide more than one response to it. Once he or she types **ask why**, however, the game will leave the conversation node after displaying the response, so there's no need for more than one response.

One further point to note is the use of double dashes (--) in the text of various responses. TADS will automatically convert each pair of dashes into one long dash, which looks better in the output than a short dash.

Finally, note that normally a Conversation Node will normally only last for one turn. That is why we needed to insert all those `<.convstay>` tags to keep this node active until the player asks the question we want asked. But we could change the default behaviour by setting the `ConvNode`'s `isSticky` property to `true`; in that case the Conversation Node would remain active until we explicitly left it, either by switching to another node, or by using a tag such as `<.convnode nil>` to leave the current node without entering another.[31] You might like to experiment with changing the code to use this alternative approach to check that you can get it to work.

The next thing that's likely to occur to the player is to **ask burner about ring**. There's an important story to be told here, so this guide will need to provide it's own version, but before seeing what it is, you might first like to try defining an `AskTopic` of your own to handle this. For the sake of argument, assume that Heidi asks "What happened to the ring – how did you manage to lose it?", and try devising your own answer. Then check that Joe responds to **ask burner about ring** as you intended.

[31] This is fact makes the NPC's current ConvNode nil – i.e. tells the program that the NPC is no longer in a Conversation Node – but only because there probably isn't a Conversation Node with 'nil' as its name; any non-existent ConvNode name would have done as well, but 'nil' serves best to make the purpose clear.

The answer we actually need here, since it's important to our plot, may be supplied thus (this time nested inside `burnerTalking` again, so put it just after the definition of `++ AskTopic @burner`):

```
++ AskTellTopic, StopEventList @ring
   [
     '<q>What happened to the ring -- how did you manage to lose it?</q> you
       ask.<.p>
     <q>You wouldn\'t believe it.</q> he shakes his head again, <q>I took it
       out to take a look at it a couple of hours back, and then dropped the
       thing. Before I could pick it up again, blow me if a thieving little
       magpie didn\'t flutter down and fly off with it!</q>',
     '<q>Where do you think the ring could have gone?</q> you wonder.<.p>
     <q>I suppose it\'s fetched up in some nest somewhere,</q> he sighs,
     <q>Goodness knows how I\'ll ever get it back again!</q>',
     '<q>Would you like me to try to find your ring for you?</q> you
       volunteer earnestly.<.p>
     <q>Yes, sure, that would be wonderful.</q> he agrees, without sounding
      in the least convinced that you\'ll be able to. '
   ]
;
```

There's one problem with this (actually, there's two, but we'll come to the second one in a minute). This part of the conversation presupposes that Joe has told Heidi the sad story of how he came to lose the ring, but that she hasn't found it yet. Although events could happen that way round, it's perfectly possible that the player could locate the ring before getting into conversation with Joe. That's easy enough to test for – if the ring has been found `gPlayerChar.hasSeen(ring)` will be true, otherwise it will be `nil`. But where do we put this test without a lot of clumsy coding? Once again TADS 3 comes to our rescue with a very neat solution, we simply use an `AltTopic` (directly after the `AskTellTopic` we've just defined):

```
+++ AltTopic
    "<q>I found a ring in a bird's nest, up a tree just down there.</q> you
     tell him, pointing vaguely southwards, <q>Could it be yours?</q><.p>
    <q>Really?</q> he asks, his eyes lighting up with disbelieving hope,
     <q>Let me see it!</q>"
    isActive = (gPlayerChar.hasSeen(ring))
;
```

There's no need to code either the commands or the object the `AltTopic` is to respond to, it will respond to whatever the `TopicEntry` it's contained in responds to, provided that its `isActive` property is `true`. If its `isActive` property is `true` then it will be used in preference to the `TopicEntry` in which it is contained. In this case we want the `AltTopic` to be used instead of its main `AskTellTopic` if (and only if) Heidi has actually found the ring, which we achieve with the line `isActive = (gPlayerChar.hasSeen(ring))`. Incidentally, this is why we made the topic an `AskTellTopic` instead of simply an `AskTopic`; once Heidi has found the ring the player is just as likely to **tell burner about ring** as **ask burner about ring**.

If you now compile and run the game, you should soon encounter the second problem. If Heidi has found the ring when she asks or tells Joe about it, everything should work as expected, but if she hasn't, then asking (or telling) the charcoal burner about the ring works just like asking or telling him about a topic we haven't defined. This makes sense before Heidi has got Joe to tell his sorry tale, since she doesn't know there's a ring to ask about (which is why the library handles it this way), but once Joe has mentioned his ring she ought to be able to ask about it.

The TADS library keeps track of which things an actor knows about through objects' `isKnown` property, which should be tested through actors' `actor.knowsAbout(obj)` method. Actually, the standard library only keeps track of what the Player Character knows about by this means, but provides the `actors` parameter in case some brave soul wants to expand the system to a tracking NPCs' knowledge as well. By default `actor.knowsAbout(obj)` is true either if `obj` has been seen by the Player Character or if `obj.isKnown` has been set to `true`. The correct way of achieving the latter is by calling `gPlayerChar.setKnowsAbout(obj)`. This seems rather long-winded for what is likely to be a quite commonly needed operation, so the library offers an abbreviated form (a macro) `gSetKnown(obj)`. This macro definition is a preprocessor directive that means roughly 'whenever you see `gSetKnown(obj)` in the source code, replace it with `gPlayerChar.setKnowsAbout(obj)` before presenting it to the compiler, where `obj` can be any object name we care to use'. In other words, all we have to do to fix things is to add `<<gSetKnown(ring)>>` to the end of the output string of the appropriate SpecialTopic, thus:

```
++ SpecialTopic 'ask why' ['ask','why']
  "<q>Why will your name be mud?</q> you want to know.<.p>
  He shakes his head, lets out an enormous sigh, and replies,
  <q>I was going to give her the ring tonight -- her engagement ring --
  only I've gone and lost it. Cost me two months' wages it did. And
  now she'll never speak to me again,</q> he concludes, with another
  mournful shake of the head, <q>never.</q><<gSetKnown(ring)>>"
;
```

Now, once Joe has mentioned the ring, Heidi will be able to ask about it and get a sensible response, even is she hasn't found the ring yet. If you recompile and play the game with these changes, you should find it all works properly.

Well, not quite all, perhaps. Although the charcoal burner has told Heidi that his name is 'Joe Black', he continues to be described as 'the charcoal burner'. Not only that, but the parser refuses to recognize him if we try to refer to **joe**, **black** or **joe black**. We'll fix these problems in the next section.

5. *What's in a Name?*

Once the charcoal burner has revealed his name, we want three things to happen. First, we want his short name (the one that's displayed in room descriptions) to change from 'the charcoal burner' to 'Joe Black'; second we want the program to treat the name as a proper name, so we don't get messages like 'The Joe Black is holding a spade' or 'You see a Joe Black here'; and third, we want the parser to recognize **joe**, **joe black** or **black** as referring to Joe. The first two steps are straightforward. The third is a little more complicated.

To carry out the first two steps we simply need to execute:

```
isProperName = true;
name = properName;
```

You might think that you could achieve the third by adding 'Joe Black' to the burner's `vocabWords` property somehow, but it's not quite that easy. What we actually need to do is to add them to the game's dictionary. To add a word to the dictionary we need to call `cmdDict.addWord(obj, word, &wordtype)` where `obj` is the object we want this word to apply to, and `wordtype` is the type of word it is (adjective or noun). Moreover, we can't simply add the contents of the `properName` property into the dictionary – it must be added word by word (token by token), not as a complete phrase. Fortunately the library provides a means of extracting the individual tokens from a string; we need to use `Tokenizer.tokenize(properName)`, which returns a list of tokens. Assuming we assign this list to a local variable `tokList`, we can get at the parsed token at position i (in a form suitable for adding to the dictionary) by calling `getTokVal(tokList[i])`. But just when you thought it was getting far too complicated to follow, there's another complication. Logically, all proper names are nouns, so we should add them to the dictionary as nouns. But if we add them all to the dictionary as nouns the parser will recognize **joe** or **black** as referring to Joe, but not **joe black**, since it firmly believes that a noun phrase should only contain one noun.[32] It will thus accept **joe** and **black** as alternatives, but not both together. There is an extremely fiendish way of getting round this by defining another Grammar Production, but that's way beyond the scope of this Getting Started guide, so we'll settle for a hack instead, and that is to add every name but the last into the library as an adjective as well as a noun (which means that the parser will quite happily recognize **joe black** even though it does so on the erroneous basis of believing that **joe** is an adjective qualifying **black** – if one can talk about the beliefs of a parser).

Since finding out an actor's name some way through a game is a situation that could arise quite often, it makes sense to make all this happen as a modification of the `Actor` class (which will then work for everything of class `Actor` or one of is subclasses) rather than something specific to the burner object. The appropriate code then looks like this:

```
modify Actor
  makeProper
  {
    if(isProperName == nil && properName != nil)
    {
      isProperName = true;
      name = properName;
      local tokList = Tokenizer.tokenize(properName);
      for (local i = 1, local cnt = tokList.length() ; i <= cnt ; ++i)
       {
        if(i < cnt)
          cmdDict.addWord(self, getTokVal(tokList[i]), &adjective);
        cmdDict.addWord(self, getTokVal(tokList[i]), &noun);
       }
    }
  }
;
```

The purpose of the check `if(isProperName == nil && properName != nil)` is to stop the `makeProper` method doing anything either if the actor has already been defined as having a proper name (perhaps through a previous call to `makeProper`) or if the actor has no `properName` property defined.

Apart from `cmdDict.addWord()`, the only thing that might be really unfamiliar here is the `for` loop towards the end of the makeProper method. If you didn't read about it in chapter 1 (or you did but you've now forgotten exactly how it works), now might be a good time to refer back to p. 25 to see how it works.

[32] The exception to this is in expressions like "pile of leaves"; if an object is defined with both 'pile' and 'leaves' as its nouns, it will respond to 'pile', 'leaves' or 'pile of leaves'.

You should copy the above code into your source file (perhaps near the top after the definition of the `endGame` function) and check that it works. And then I'll confess that all along there was a somewhat simpler way we could have achieved almost the same effect without either that complicated for loop or `cmdDict.addWord`. Instead we could have replaced all the code after `name = properName;` (apart from the necessary closing braces) with `initializeVocabWith(properName);`, and it would have worked just as well. The only difference is that 'joe' would have been entered into the dictionary only as an adjective, and not also as a noun, but in practice that almost certainly doesn't matter. You may want to test this out.

But you can't test it out just yet: for either method to actually do anything in our program we need to call it somewhere. We can do this by simply adding `<<burner.makeProper>>` in the response string of `AskTopic @burner`, perhaps just after `<.convnode burner-mud>`.

There's just one more job we need to do before we can leave Joe Black. As things stand at the moment, if the player asks Joe about himself a second time, he'll still introduce himself the same way, which we obviously don't want. We could fix this by using a `EventList`, since although `<<burner.makeProper>>` won't work inside a single-quoted string, we can get round this by using a function within the `EventList`;[33] but rather than introduce that complication right now, we'll simply use another `AltTopic`. Since the burner's `isProperName` property is `nil` before he introduces himself and true afterwards (thanks to `<<burner.makeProper>>`), we can use `burner.isProperName` as the test in the `isActive` property of the `AltTopic`. The `AskTopic` and `AltTopic` then look like this:

```
++ AskTopic @burner
   "<q>My name's Heidi.</q> you announce. <q>What's yours?</q><.p>
   <q><<burner.properName>>,</q> he replies, <q>Mind you, it'll soon be
    mud.</q> <.convnode burner-mud><<burner.makeProper>>"
;

+++ AltTopic, StopEventList
  [
   '<q>Have you been a charcoal burner long?</q> you ask.<.p>
   <q>About ten years.</q> he replies. ',
   '<q>Do you like being a charcoal burner?</q> you wonder, <q>It seems
   rather messy!</q><.p>
   <q>It\'s better than being cooped up in some office or factory all day,
   at any rate.</q> he tells you. ',
   '<q>What do you do when you\'re not burning charcoal?</q> you
     enquire.<.p>
   <q>Oh -- this and that.</q> he shrugs. '
  ]
  isActive = (burner.isProperName)
;
```

One further refinement you could add, which I'll leave as an optional exercise for the reader, is to add `SuggestedAskTopic` to the class list of each of the main topic entries in the `burnerTalking` state. If you do that you'll need to add a name property to each of the `AskTopic` definitions, perhaps `name = 'the smoke'`, `name = 'the fire'`, `name = 'the ring'` and `name = 'himself'` as appropriate. The player can then use the **topics** command to see what topics Joe is likely to respond to.

[33] Syntactically this looks quite straightforward; we can use the construction `{: "double-quoted string" }` inside an EventList anywhere we could have used a single-quoted string, but it's a bit complicated to explain *why* this works at this point.

Another optional exercise you might like to try is expanding the range of topics to which Joe will respond meaningfully. You can do this with a mix of `AskTopics`, `TellTopics`, `AskTellTopics`, `GiveTopics`, `ShowTopics` and `GiveShowTopics` as the mood takes you. You might also want to replace the `DefaultAskTellTopic` with a separate `DefaultAskTopic` and `DefaultTellTopic`. You're not restricted to having Joe talk about objects defined in the game. If, for example, you think he should have an opinion on Oxford Blue (a type of cheese), you could define an Oxford Blue topic using the Topic class:

```
tOxfordBlue: Topic 'oxford blue/cheese';
```

The `Topic` class can be used for any concrete or abstract topic not otherwise represented in the game. Unlike game objects (i.e. those derived from `Thing`, which `Topic` isn't), all `Topics` start out known to the player character unless you define otherwise, which means that Heidi doesn't actually have to have encountered any Oxford Blue cheese in the game in order to ask Joe about it. Incidentally, there's nothing magic about the `t` at the start of the object name here (`tOxfordBlue`), it's just a convention I use to distinguish `Topics` from other types of object.

One further feature you may want to try out is the `<.reveal>` tag, which you can use to keep track of what's already been said. This works by keeping track of a list of arbitrary strings (or 'keys') that have been revealed, either through the `gReveal()` macro, for example `gReveal('foo')`, or through a `<.reveal foo>` tag inserted into a string (either single-quoted or double-quoted). You can then test whether the key has been revealed using `gRevealed()`, e.g. in a declaration like `isActive = gRevealed('foo')` on an `AskTopic`. For example, at the end of Joe's reply to Heidi's question on Oxford Blue cheese, you might append a `<.reveal oxford-blue>` tag so that other `AskTopics` or `AltTopics` can test whether this part of the conversation has taken place.

At this point you might like to experiment with increasing Joe's conversational range before moving on to the next chapter. If you want to be particularly adventurous, after trying out a few `AskTopics` and `TellTopics`, you could try adding some `AltTopics` and maybe even the odd extra `ConvNode` or two, complete with some more `SpecialTopics`.

Our game has now reached a point at which, barring full testing, adding in more decoration objects and the like, it could be regarded as complete. It can be played through from start to finish, our NPC provides a reasonably good explanation of the plot (well, the bit about the magpie may be a bit far-fetched, but if Rossini can get away with it I don't see why we shouldn't), and there's a reasonable closure when the Heidi hands the ring over to poor old Joe. As a tutorial game, we could simply leave it there, even though it's never going to win any prizes in IF competitions. In the following chapters, however, we'll complicate things for Heidi by putting more obstacles between her and the ring, not because this will transform the game into a marvellous one (that would require a miracle), but because it will provide a convenient vehicle for introducing some further features of TADS 3.

Chapter Six - Expanding the Horizons

1. *Doors and Windows*

If you've managed to follow this *Guide* so far, you should have grasped most of the basics of programming in TADS 3. In the present chapter we'll look at more features of the library, but we'll move on a bit more briskly, on the assumption that much of the code should start to be self-explanatory.

In order to make Heidi's life more difficult, we'll make it harder for her to get hold of the chair she needs to climb the tree. To do that, we simply need to supply the cottage with a locked front door and hide the key in some out-of-the-way place.

The first thing to realize is that doors in TADS 3 are normally two-sided. That is, they are generally represented not by *one* object, but by *two*, the two objects being the two sides of the door. At first sight this may seem something of an unnecessary complication, and it does require a little more work, but not as much more work as you might think. Provided the two sides of the door are properly set up, the library will take care of keeping them in sync (i.e. ensuring that one side is open or closed or locked or unlocked when the other is). It will also take care of making travel through one side of the door result in the traveler arriving in the location of the other side. One reason for doing it this way (i.e. with each side of the door represented by a separate object) is that it allows more flexibility; the two sides of a door often aren't identical: they may, for example, be painted different colours, or they may use different locking mechanisms, say with one side requiring a key and the other using a paddle.

We'll start by adding the outside of the front door (which should be contained in `outsideCottage`):

```
+ cottageDoor : LockableWithKey, Door 'door' 'door'
  "It's a neat little door, painted green to match the window frame. "
  keyList = [cottageKey]
;
```

To make the door work, we also need to change the `in` property of `outsideCottage` to read `cottageDoor` instead of `insideCottage`. As noted above, we also need to define the other side of the door:

```
+ cottageDoorInside : Lockable, Door -> cottageDoor 'door' 'door';
```

This needs to be located in `insideCottage`, and we need to change the `out` property of `insideCottage` to `cottageDoorInside`. The `->cottageDoor` is a template shortcut for assigning `cottageDoor` to the `masterObject` property, which (a) keeps both sides of the door in sync (both open/closed and locked/unlocked) together, and (b) tells each door what its other side is (through code executed in the preinitialization routine, which we don't need to worry about).

Note that we have defined the outside of the door as a `LockableWithKey` and the inside as simply `Lockable`; this reflects the way many house doors in fact behave (we don't need a key to lock or unlock the front door from the inside). Note also that the *order* of the classes here is important: `Lockable`, `LockableWithKey` or `IndirectLockable` must come *before* Door in this kind of declaration, or else the lock won't work. The short explanation for this is that `Lockable`, `LockableWithKey` and

`IndirectLockable` are examples of *mix-in* classes (not derived from Thing) which must come before the any Thing-derived class with which they are combined.[34]

Before this door will work, we have to define the key object. As a temporary measure (we'll move it elsewhere later), we'll do this with simply:

```
cottageKey : Key 'small brass key' 'small brass key' @outsideCottage;
```

Since Heidi's now locked out of the cottage (or would be if the key was not so readily in reach), an obvious thing for her to try is looking through the window to see what's inside. It's probably a good thing to allow this, since if she can see that the chair is there it will make it all the more obvious that it's worth going to look for the key.

The question is, how should this be implemented? We could just write a LookThrough routine that displayed a pre-programmed message, but that's less than ideal, since the contents of the cottage could change as the player moves objects around. Writing a LookThrough routine that does the job properly is quite tricky, so for now we'll attempt something a little less ambitious: a window through which whatever is on the other side is visible. We'll return to a more sophisticated LookThrough later.

To create a window through which the contents of another location are visible, we need to use a `SenseConnector`, and locate it in the two rooms joined by the window:

```
cottageWindow : SenseConnector, Fixture 'window' 'window'
  "The cottage window has a freshly painted green frame.
    The glass has been freshly cleaned. "
   connectorMaterial = glass
  locationList = [outsideCottage, insideCottage]
;
```

Since `SenseConnector` is a `MultiLoc` (an object that exists in more than one location) we do not define its `location` property; instead we define its `locationList` to contain a list of the locations that contain the window, in this case the inside and outside of the cottage. The other important property to define here is `connectorMaterial`; the TADS 3 library defines a number of different materials that transmit the various senses in different ways. Glass is defined to be transparent to sight, but opaque to sound, smell and touch (the fact that real world glass may allow sound to pass need not concern us here, since in this case sight is the only sense we're worried about). This means that from outside the cottage the player character will be able to see anything located inside, and from inside, the player character will be able to see anything left directly outside, but that Heidi will not be able to smell, hear or touch anything through the window.

If you compile and test the game now, you'll find that this works, but that objects visible through the window are listed in a less than ideal fashion. There are several steps we can take to improve that. You'll recall that we defined an `initSpecialDesc` on the chair. The first problem is that we'll now see that `initSpecialDesc` when Heidi is standing just outside the cottage:

[34] For the long explanation, see the article on Multiple Inheritance in the *Technical Manual*.

In front of a cottage
You stand just outside a cottage; the forest stretches east. A short path leads round the cottage to the northwest.

A plain wooden chair sits in the corner.

This description is plainly inappropriate when the chair is being viewed through the window from outside. What we want is a different type of `initSpecialDesc` that's shown when the chair is viewed from a room other than the one in which it's actually located; for that we use `remoteInitSpecialDesc`. Add the following to the definition of the chair object:

```
remoteInitSpecialDesc(actor) { "Through the cottage window you can
   see a plain wooden chair sitting in the corner of the front room. "; }
```

The `actor` parameter refers to the actor doing the looking, normally the player character. The parameter can be used to test where the chair is being viewed from; for example, if there were a second window looking into the room from the cottageGarden (we'll be this garden adding later, though not the second window), your `remoteInitSpecialDesc(actor)` routine could test for `actor.isIn(cottageGarden)` and `actor.isIn(outsideCottage)` and provide different descriptions for the two cases. In this case this is unnecessary, however, since `outsideRoom` is the only remote location from which the chair can ever be viewed.

If you test the game now, you'll find the window works okay with the chair, but is not so good with portable objects. For example, if you drop the key outside the cottage and then go inside, you'll see the key listed as:

In the in front of a cottage, you see a small brass key.

Similarly if you leave the key inside the cottage and then go back outside, you'll find the key listed as:

In the inside cottage, you see a small brass key.

The library provides two ways to fix this: (1) you can give a room an `inRoomName`, which is the name to be used when an item is listed as being in that room when viewed from another location; or (2) you can define a custom `remoteRoomContentsLister` which defines how portable items will be listed when viewed in a remote location. If you use method (1) you define `inRoomName` on the room that's being viewed remotely; if you use method (2) you define the `remoteRoomContentsLister` on the room from which the remote viewing is taking place. In order to illustrate both methods we'll use method (1) for looking in through the window from the outside, and method (2) for looking out through the window from the inside. This means that both methods need to be implemented on `insideCottage`:

```
insideCottage : Room 'Inside Cottage'
  "The front parlour of the little cottage looks impeccably neat.
  The door out is to the east. "
  out = cottageDoorInside
  east asExit(out)
  inRoomName(pov) { return 'inside the cottage'; }
  remoteRoomContentsLister(other)
```

```
    {
      return new CustomRoomLister('Through the window, {you/he} see{s}',
            ' lying on the ground.');
    }
;
```

The `pov` parameter (in `inRoomName`) represents the point of view, and could be used to give a room different names depending on where it was being viewed from (e.g. on a long stretch of road you might want a particular stretch of road named as 'on the road to the south' when the pov is north of it and 'on the road to the north' when the pov is to the south of it). The `other` parameter of `remoteRoomContentsLister` is the other location that's being viewed from here, which would allow you to vary the lister according to which room's contents was being described; in this case the only other location visible from `insideCottage` is `outsideCottage`, so it's not necessary to make use of this parameter. The two parameters supplied to new `CustomRoomLister` are the prefix and suffix strings. This will result in message like:

Through the window, you see a small brass key lying on the ground.

Similarly, if the key is left inside the cottage, then from the outside you'd see:

Inside the cottage, you see a small brass key.

Both of these messages are substantial improvements over what we had before. We have still not implemented a response to explicitly looking through the window, but since this is rather trickier, we'll leave it till later.

2. *Crossing the Stream*

As the next step to making things more complicated for Heidi, we'll put the key in a field on the far side of a stream. First we need to add two extra locations to accommodate the stream:

```
pathByStream : OutdoorRoom 'By a stream'
  "The path through the trees from the southeast comes to an end on
  the banks of a stream. Across the stream to the west you can see
  an open meadow. "
  southeast = fireClearing
  west = streamWade
;

streamWade : RoomConnector
  room1 = pathByStream
  room2 = meadow
;

meadow : OutdoorRoom 'Large Meadow'
  "This large, open meadow stretches almost as far as you can see
  to north, west, and south, but is bordered by a fast-flowing stream
  to the east. "
  east = streamWade
;
```

The reason for using the separate `RoomConnector` object, `streamWade`, will gradually become apparent. At the moment note that it simply connects the room in its `room1` property to the room in its `room2` property. It also furnishes an example of how we can set the direction property of a room to an explicit connector object. One further thing we need to do at this stage is to set the `northwest` property of `fireClearing` to `pathByStream`.

Next we'll move the small brass key to the meadow and tweak its properties a little.

```
+ cottageKey : Key 'small brass brassy key/object/something' 'object'
  "It's a small brass key, with a faded tag you can no longer read. "
  initSpecialDesc = "A small brass object lies in the grass. "
  remoteInitSpecialDesc(actor)
  {
    "There is a momentary glint of something brassy as
    the sun reflects off something lying in the meadow across the stream. ";
  }
  dobjFor(Take)
  {
    action()
    {
      if(!moved)
         addToScore(1, 'retrieving the key');
      inherited;
      name = 'small brass key';
    }
  }
;
```

The reason for the special `dobjFor(Take)` routine is that if we let the key start with the name 'small brass key', it might give its presence away prematurely, so we give it about the vaguest name we can in its initial definition and then change it to a more meaningful name when it's picked up. Note that we have once again used `remoteInitSpecialDesc`, which (once we've done some more clever stuff) will be the description that's displayed when we view the key from a distance, in this case the other side of the stream. Note that this is a method, not a property, and it takes a single parameter `pov` (point of view). This parameter represents the actor who is doing the looking, and would allow you to alter the message displayed depending on where the actor was (e.g by testing for `if(pov.isIn(pathByStream))`). In this case the test is unnecessary, since there is only one location from which the key can be viewed remotely before it is moved.

Now comes the clever stuff. In order to make objects in room A visible from room B we need to join the two locations together with a `DistanceConnector`; which is particular kind of `SenseConnector` (which we met before in connection with the cottage window); a `SenseConnector` can exist in two or more locations since it is a subclass of `MultiLoc` (more of which anon). A `DistanceConnector` has a library template that makes it exceedingly easy to define; all we need to add is:

```
DistanceConnector [pathByStream, meadow];
```

The list in square brackets is in fact the `locationList` property, the name of which should be fairly self-explanatory. Note that `DistanceConnector` is a descendant of both `MultiLoc`, which is a mix-in class, and `Intangible` (since the connector has no physical presence). Another `MultiLoc` object we could use here would be a stream, which runs through both the rooms. And while we're at it, we'll make it possible for the player to cross the stream with the command **cross stream**.

```
stream : MultiLoc, Fixture 'stream' 'stream'
  "The stream is not terribly deep at this point, though it's flowing
```

```
    quite rapidly towards the south. "
  locationList = [pathByStream, meadow]
  dobjFor(Cross)
  {
    verify() {}
    check() {}
    action()
    {
      replaceAction(TravelVia, streamWade);
    }
  }
;
```

Note that being a `MultiLoc` object (like the `DistanceConnector`), the stream does not have a `location` property (its list of locations instead being held in `locationList`). Note also that for this to work properly, `MultiLoc` must come first in the class list; `MultiLoc` is a mix-in class that should always be combined with something else.

Part of the value of defining a separate `streamWade` object now becomes apparent; it makes the coding of the action method of `dobjFor(Cross)` exceedingly simple. Instead of having to test for which side of the stream we're on to decide which side we need to end up on when we cross the stream, we simply TravelVia `streamWade` and leave `streamWade` to sort it all out. But as we'll see shortly, that's only part of the story.

In the meantime, there's another little matter we need to attend to. Unlike the other verbs we've used so far, there's no definition of Cross anywhere in the TADS 3 library, so we have to create our own. For details of how to do this in general, see the *Technical Manual* (but there's no need to consult it right now – you can finish this guide first). Here we'll just list the steps for this simple case.

First, we need to define both `CrossAction` and its associated grammar. A couple of library macros hide most of the complication of all this, and all we need write is:

```
DefineTAction(Cross);

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

We use the `DefineTAction()` macro to define a Transitive Action (hence `TAction`), which means an action taking a direct object (as opposed to an `IAction` like **Look** which takes no objects, or a `TIAction` like **put x on y** which takes both a direct object and an indirect object). We next use the `VerbRule()` macro to define the grammar for the command, that is the form of words that a player can use to invoke it.

The name of the `VerbRule` (here `Cross`) can be anything we like, so long as it's unique among the `VerbRule` names in our game. It doesn't actually *need* to match the name of our action, it's just (a) a convenient way of ensuring a unique `VerbRule` name and (b) an obvious way of making it clear what the `VerbRule` is for. After naming the `VerbRule` we next need to define its grammar, i.e. the phrase that the player must enter to invoke this command. This will normally consist of a fixed element, such as the name of the verb, in this case 'cross', followed by a placeholder for the noun or nouns that the player wants the command to apply to. For a `TAction` this placeholder can either be `singleDobj` (meaning that only one direct object is allowed) or `dobjList` (meaning that the command can be applied to several direct objects at once, as in **take the red ball, the long stick, and the bent banana**).

It would make no sense to cross several objects at once, so we definitely want `singleDobj` rather than `dobjList` here. We could, if we wanted, have defined more synonyms for the verb, e.g. `('cross' | 'ford') singleDobj`, but once more I'll leave that as an exercise for the interested reader. The point to note is that if we do want to define alternative phrasings, we use a vertical bar (`|`) to separate the alternatives, and brackets to group them. The brackets would be necessary in the foregoing example, since without them we'd have `'cross' | 'ford' singleDobj`, which would mean 'cross' or 'ford something', rather than 'cross something' or 'ford something', as we'd actually want.

After the definition of the grammar for the command comes a colon followed by the name of the action class, which is the name we gave the action plus the word 'Action' appended, hence `CrossAction`. If you think this looks rather like declaring our `VerbRule` (strictly speaking, our grammar definition) to be of class `CrossAction`, then you're right; but again this isn't an issue that need concern us here, beyond noting that the `DefineTAction(Cross)` macro in fact defines a new class called `CrossAction` as a subclass of `TAction`.

We then have to define a `verbPhrase` so that the parser can construct certain messages, such as '(first crossing the stream)' or 'What do you want to cross?' if it needs to. The correct format for a verb phrase for a `TAction` should be reasonably clear from the example shown: first the infinitive (without 'to') followed by the present participle with a slash (oblique) in between (hence 'cross/crossing'). Then a placeholder for a direct object, enclosed in brackets (hence '(what)'). Note that this placeholder may be used by the parser to construct a question about a missing direct object ('What do you want to cross?'), so for verbs that were more likely to be applied to people (e.g. 'thank') you'd want to use '(who)' or, even more correctly, '(whom)' rather than '(what)'.

One more step we have to take is to define what happens when **cross** is used with any noun other than the stream, which we can do by modifying the definition of the Thing class:

```
modify Thing
  dobjFor(Cross)
  {
   verify()
   {
      illogical('{The dobj/he} {is} not something you can cross. ' );
   }
  }
;
```

Note here how we've begun the illogical response with `'{The dobj/he} {is}'` rather than `'{The dobj/he} is'`. By putting 'is' in curly braces we ensure that it will always agree in number with the name of the direct object (which is what, of course, `'{The dobj/he}'` expands to). This ensures that if the direct object were, say, some flowers growing on the river bank, then **cross flowers** will respond with 'The flowers are not something you can cross' rather than the incorrect 'The flowers is not something you can cross'.

If you now compile and run the game it should all work, though getting across the stream doesn't seem to be much of a puzzle. We can make it more of one if Heidi has to wear a pair of old boots before she can cross. To start with we'll leave the boots lying by the side of the stream. Then all we have to do is to modify the `streamWade` object so that it only allows anyone to pass when they're wearing the boots.

Before looking at the solution below, you may like to try to work out how to do all this yourself. The only new thing about the boots is that we need to make them of class `Wearable`, so Heidi can put them on. The trick is then to work out how to prevent Heidi from crossing the stream unless she is wearing the boots. You should be able to work it out by analogy from the way we prevented Heidi from climbing the tree unless she's standing on the chair.

First here's the boots; as noted above the only thing new about them is that we make them of class `Wearable`, so Heidi can put them on:

```
boots : Wearable 'old wellington pair/boots/wellies' 'old pair of boots'
  @pathByStream
 "They look ancient, battered, and scuffed, but probably still waterproof. "
;
```

Next we need to modify the `RoomConnector` so that Heidi can only cross the stream when she's wearing the boots:

```
streamWade : RoomConnector
  room1 = pathByStream
  room2 = meadow
  canTravelerPass(traveler) { return boots.isWornBy(traveler); }
  explainTravelBarrier(traveler)
  {
    "Your shoes aren't waterproof. If you wade across you'll get your feet
      wet and probably catch your death of cold. ";
  }
;
```

And that's all there is to it. If you try the game again you'll find you can't cross the stream (in either direction) unless you're wearing the boots. The next job is to hide the boots in a less obvious place.

3. *Burying the Boots*

We'll hide the boots by burying them in a cave and then provide a means of digging them out again. In the next chapter we'll give the cave a dark interior so we can look at the handling of light sources, but so as not to handle too many new problems at once, we'll leave that to one side for now, and concentrate on giving the cave a floor that can be excavated.

But first we have to add the cave and a means of getting to it. Again, you may like to try doing this yourself rather than just copying the code overleaf. Create a room to the south of `forest` called `outsideCave`, and give it an appropriate name and description. Then create a room called `insideCave` (representing the inside of the cave) to the south of that. Be sure to implement all the appropriate connections (including one south from `forest`!). Also, give some thought to which class is most appropriate to each of these two new locations, and also whether it may be appropriate to use the `asExit()` macro to allow alternative commands for moving between them.

Once you've done that and checked that it all works, you'll probably have an `outsideCave` location that mentions a cave somewhere in the description – at least, you certainly should do. So what if the player types the command **enter cave**? You'd better add another object to handle that.

Finally, if you're feeling really adventurous, you could try to devise a way of burying the boots in the cave so that they are only discovered when Heidi digs in the ground with a spade (which you'll need to provide). For inspiration, you could look back at the way we hid the ring in the nest, or the stick in the pile of twigs.

Here's one way of implementing the new rooms:

```
outsideCave : OutdoorRoom 'Just Outside a Cave'
  "The path through the trees from the north comes to an end
  just outside the mouth of a large cave to the south. Behind the cave
  rises a sheer wall of rock. "
  north = forest
  in = insideCave
  south asExit(in)
;

+ Enterable 'cave/entrance' 'cave'
  "The entrance to the cave looks quite narrow, but probably just wide
  enough for someone to squeeze through. "
  connector = insideCave
;

insideCave : Room 'Inside a large cave'
  "The cave is larger than its narrow entrance might lead one to expect.
    Even a tall adult could stand here quite comfortably, and the cave
    stretches back quite some way. "
  out = outsideCave
  north asExit(out)
;
```

There's nothing that requires comment here, apart from the need to add `south = outsideCave` to the definition of the `forest` room (whose description already mentions a path to the south that might have seemed a bit superfluous up to now.)

Now let's make a number of assumptions about how we want to handle the action of digging. The library provides both Dig and DigWith verbs (e.g. **dig ground** and **dig ground with spade**). Let's assume that in order to be able to dig, it's necessary to be holding a spade, and that there's only one spade object in the game. Let's further assume that although we should allow the player to **dig ground with spade**, it's unnecessarily pedantic to insist on that form of command and refuse to respond to **dig ground** when the spade is being held (if someone's holding a spade it's pretty obvious that's what they want to dig with). Finally, let's assume that there may in general be more than one place where we might want the player to be able to dig, so that it would be useful to define a Diggable class that can handle all this. The class might then look like this:

```
class Diggable : Floor
  dobjFor(DigWith)
  {
    preCond = [objVisible, touchObj]
    verify() {}
    check() {}
    action()
    {
      "Digging here reveals nothing of interest. ";
    }
  }
;
```

At first sight, this class definition may look a little surprising, since we have done nothing to handle the case where the player simply types **dig ground**. In fact this is already catered for by the library, which defines the `action` method of Dig on the `Thing` class as: `action() { askForIobj(DigWith); }`. This more or less does what it looks like: if the player types **dig whatever** without specifying an indirect object, (i.e. an implement to dig with) the game will respond with "What do you want to dig in it with?". If the player then types the name of an implement, such as "spoon", the game will treat the whole command as if it were **dig in whatever with spoon**. On the other hand, if one and only one suitable digging implement is to hand, then the parser will automatically assume that's what the player wants to use. In this game the only suitable digging implement is the spade, so if the player types **dig in ground** when Heidi is already holding the spade, the parser will automatically select the spade and treat the command as if it had been **dig in ground with spade**; this is precisely what we want. Normally we'll override the `dobjFor(DigWith) action()` method on the specific object to provide a particular response, but we provide a default response on the class.

In order to dig in something you must be able to touch it. In practice, you probably need to be able to see it as well. We take care of enforcing these conditions with the line `preCond = [objVisible, touchObj]`, which adds a couple of *preconditions* to the digging action on the digging class. Although it's possible (and not actually all that difficult) to define preconditions of your own, the common ones are already defined for you in the library. In particular, the `objVisible` precondition prevents the action from proceeding if the object is not visible for any reason (this will become relevant when we go on to make the cave dark). Similarly `touchObj` will not allow an actor to dig in the ground unless the actor is in a position to touch the ground (this precondition will not strictly affect anything in this game, but we'll add it anyway for the sake of completeness and in order to illustrate the principle).

We next need to define the spade:

```
spade : Thing 'sturdy spade' 'spade'
@insideCave
  "It's a sturdy spade with a broad steel blade and a firm wooden handle. "
  initSpecialDesc = "A sturdy spade leans against the wall of the cave. "
  iobjFor(DigWith)
  {
    verify() {}
    check() {}
  }
;
```

Placing the spade inside the cave is a temporary measure to make it easy to test that the digging operation works as we intend. The only point to note about the definition of this object is the empty `verify()` and `check()` methods we supply for `iobjFor(DigWith)` to ensure that the spade raises no objection to be used as a digging implement (i.e. the indirect object of a **dig with** command).

Now we need to supply our `Diggable` object, the ground. Since digging the ground will create a hole, and a pair of boots will be found lurking in the hole, we may as well deal with them at the same time.

```
caveFloor : Diggable 'cave floor/ground' 'cave floor'
  @insideCave
  "The floor of the cave is quite sandy; near the centre is
  <<hasBeenDug ? 'a freshly dug hole' : 'a patch that looks as if it has
  been recently disturbed'>>. "
  hasBeenDug = nil
  dobjFor(DigWith)
  {
    check()
    {
      if(hasBeenDug)
      {
        "You've already dug a hole here. ";
        exit;
      }
    }
   action()
   {
     hasBeenDug = true;
     "You dig a small hole in the sandy floor and find a buried pair of
     old Wellington boots. ";
     hole.moveInto(self);
     addToScore(1, 'finding the boots');
   }
  }
;

hole : Container, Fixture 'hole' 'hole'
  "There's a small round hole, freshly dug in the floor near the centre
  of the cave. "
;

+ boots : Wearable 'old pair of wellington boots/wellies' 'old pair of
boots'
 "They look ancient, battered, and scuffed, but probably still waterproof. "
  initSpecialDesc = "A pair of old Wellington boots lies in the hole. "
;
```

Again, there's little that should require much explanation here. Note that we have moved the original boots and put them inside the hole, giving them an appropriate `initSpecialDesc`. Since the act of digging undoubtedly will be to create a hole, we make the creation of the hole (simulated by moving the hole object into the floor) the main effect of the DigWith action – again note that we do this by using the hole's `moveInto` method, *not* by setting its `location` property directly. We make the hole both a `Container` (so the boots can be in it) and a `Fixture` (so we can't carry it away). We use the `check()` method to trap a second or subsequent attempt to dig in the floor, although it would have worked just as well to put the same test in the `action()` method – in this case it's simply a matter of preference (I slightly prefer it the way I did it because the message displayed in the `check` method implies that a second or subsequent request to dig in the cave floor is not even attempted, so there should be no action notifications). The `desc` property of the floor makes use of the double angle brackets, the `?:` ternary operator and the custom `hasBeenDug` property to display an appropriate description.

Note that as of TADS 3.1.0 there is another way we could have written the desc property of the cave, namely:

```
caveFloor : Diggable 'cave floor/ground' 'cave floor'
  @insideCave
  "The floor of the cave is quite sandy; near the centre is a
  <<if hasBeenDug>>freshly dug hole<<else>>patch that looks as if it has
  been recently disturbed<<end>>. "
```

Either way, this *almost* works fine, apart from one thing: as you'll no doubt discover, it you haven't tried it already, when you try to **dig floor with spade** you'll be greeted with the message:

Which floor do you mean, the cave floor, or the floor?

This is somewhat annoying, to say the least. The reason for it is that the `Room` class defines a default set of room components: four walls, a floor, and a ceiling, which normally provide an uninformative default message if you try to examine them. So what we should have done, instead of using @`insideCave` to put our custom floor into the cave, was to include it in the list of room parts. While we're at it, we may as well replace some of the other default room parts:

```
caveNorthWall : DefaultWall 'north wall' 'north wall'
  "In the north wall is a narrow gap leading out of the cave. "
;

caveEastWall : DefaultWall 'east wall' 'east wall'
  "The east wall of the cave is quite smooth and has the faint remains of
   something drawn or written on it. Unfortunately it's no longer possible
   to discern whether it was once a Neolithic cave painting or an example
   of modern graffiti. "
;
```

We then override the `roomParts` property of `insideCave`. At the same time, we must be careful to remove `@insideCave` from the definition of `caveFloor`, otherwise we'll effectively be including the floor in the cave twice. While we're at it, we'll also tweak `insideCave's` description so that it includes a description of the floor:

```
insideCave : Room 'Inside a large cave'
  "The cave is larger than its narrow entrance might lead one to expect.
   Even a tall adult could stand here quite comfortable, and the cave
   stretches back quite some way. <<caveFloor.desc>>"
  out = outsideCave
  north asExit(out)
  roomParts = [caveFloor, defaultCeiling,  caveNorthWall,
               defaultSouthWall, caveEastWall, defaultWestWall]

;
```

By the way, note that we made Diggable inherit from `Floor` rather than, say, `Fixture`; this tells the library that the caveFloor (derived from `Floor` via `Diggable`) is the room part acting as the floor, so that, for example **put torch on ground** is equivalent to **drop torch**. If you compile and run the game again you should find it works much more satisfactorily, with the added bonus that if you examine the cave walls, two of them will be a bit more interesting than the defaults would have been.

4. *Calling a Spade a Spade*

Clearly leaving the spade conveniently leaning against the wall of the cave is a bit too obvious, even for a simple tutorial game such as this. It obviously needs to start life somewhere else, and in fact we've already indicated where that somewhere else must be, since we've already described the charcoal burner as wielding a spade. To obtain the spade, therefore, Heidi needs to ask Joe for it.

This may be achieved by first of all adding a suitable `AskForTopic` to the list of TopicEntries in the `burnerTalking InConversationState`. Here again this is an exercise you might like to try for yourself before turning over the page to see how this guide does it. An `AskForTopic` works just like the other types of `TopicEntry` we've seen, except that it responds to **ask for whatever** instead of, say, **ask about whatever**. You'll need to make sure that your new `AskForTopic` responds specifically to a request for the spade, and that it results not only in Joe saying Heidi can take it, but in Heidi actually acquiring it. You'll also need to handle the case where the player issues an **ask for spade** command when Heidi already has the spade. And, of course, you'll need to make some appropriate adjustments to the spade object itself, so that it starts out being carried by Joe rather than leaning against the wall of the cave.

This is how we do it here:

```
++ AskForTopic @spade
   topicResponse
   {
      "<q>Could I borrow your spade, please?</q> you ask.<.p>
      <q>All right then,</q> he agrees a little reluctantly, handing you the
       spade, <q>but make sure you bring it back.</q>";
      spade.moveInto(gActor);
   }
;
```

If you compile the game (yet again) and try all this out, you'll find that there's still a problem: even after Joe hands the spade over he's described as still leaning on it (while he's talking) or still using it (when he goes back to work). But this problem turns out to be an opportunity to show how to give Joe a slightly wider range of behaviour. The approach we'll take is to give him another pair of ActorStates which define what he does when he's without his spade. We'll assume that once he's handed over his spade he's particularly anxious to get it back, and won't discuss anything until it's been returned. The implementation relies on switching ActorStates as Joe gives the spade to Heidi and as Heidi gives it back again. The two new ActorStates may be defined as follows:

```
+ burnerFretting : InConversationState
  specialDesc = "{The burner/he} is standing talking to you with his
   hands on his hips. "
  stateDesc = "He's standing talking to you with his hands on his hips. "
  nextState = burnerWaiting
;

++ burnerWaiting : ConversationReadyState
  specialDesc = "{The burner/he} is walking round the fire, frowning as
   he keeps instinctively reaching for the spade that isn't there. "
  stateDesc = "He's walking round the fire. "
;

+++ HelloTopic
   "<q>Hello, there.</q> you say.<.p>
    <q>Hello, young lady - have you brought my spade?</q> he asks. "
```

```
;

+++ ByeTopic
    "<q>Bye, then.</q> you say.<.p>
    <q>Don't be too long with that spade - be sure to bring it right
      back!</q> he admonishes you. "
;

++ GiveShowTopic @spade
  topicResponse
  {
    "<q>Here's your spade then,</q> you say, handing it over.<.p>
    <q>Ah, thanks!</q> he replies taking it with evident relief. ";
    spade.moveInto(burner);
    burner.setCurState(burnerTalking);
  }
;

++ AskForTopic @spade
  "He doesn't have the spade. "
  isConversational = nil
;

++ AskTellTopic @spade
  "<q>This seems a very sturdy spade,</q> you remark.\b
   <q>It is -- look after it well, I need it for my work!</q>
    {the burner/he} replies. "
;

+++ AltTopic
  "<q>I seem to have left your spade somewhere,</q> you confess.\b
  <q>I hope you can find it then!</q> {the burner/he} remarks
   anxiously. "
  isActive = (!spade.isIn(burner.location))
;


++ DefaultAskTellTopic
  "<q>We can talk about that when I've got my spade back,</q>
      he tells you. "
;
```

There's only a couple of points to note here. The first is that we include an `AskForTopic` to handle the case where the player asks for the spade again when Joe's already handed it over; since Joe will always be in the `burnerFretting` state when he doesn't have his spade, we simply include this `AskForTopic` as one of the TopicEntries in that state. In this case, instead of having Joe respond we simply display a message indicating that Joe is spadeless (we add an appropriate `AskTellTopic` and `AltTopic` to handle the case in which Heidi talks about the spade while Joe is in this state). We then add `isConversational = nil` to the definition of the topic to show that this is not a conversational interchange, so no greeting protocols will be initiated by the player character asking Joe for the spade while he's in this in state.

The second is that for all this to work as expected it is, of course, necessary to relocate the spade from the cave to the burner in your code.

The third is the explicit definition of `nextState = burnerWaiting` in the `burnerFretting` state; this is necessary because we change from one `InConversationState` to another in mid-conversation, and without the explicit definition of `nextState` (which defines which `ActorState` the `Actor` is to switch to when the conversation is terminated from that `InConversationState`) the program becomes a bit confused by the mid-conversation switch of states. For the same reason we now need to add `nextState = burnerWorking` to the definition of `burnerTalking`.

The other point worth noting is the use of `setCurState(state)` to change the actor's current actor state (don't simply write something like `burner.curState = burnerTalking;`). We need to use the same method in our handling of AskFor to get Joe to switch into his `burnerFretting` state. Add the following line immediately after `spade.moveInto(gActor);` in the `topicResponse` method of the first `AskForTopic @spade`:

```
getActor().setCurState(burnerFretting);
```

Everything should now work fine, but there is one more refinement we can add, not because the game really needs it, but because it allows us to try out an aspect of TADS 3 NPC programming we haven't seen yet. So far, the player has taken all the initiative in starting a conversation; in TADS 3 it's possible to make an NPC initiate a conversation. In this game, we'll make Joe so anxious to get his spade back that every time Heidi walks into his clearing he'll ask for it (until he gets it back), without waiting for her to address him first. We do this using his `initiateConversation(state, 'name')` method, where state is the name of the `ActorState` (normally an `InConversationState`) we want him to switch into, and 'name' is the name of a Conversation Node we want activated (as the NPC's way of initiating the conversation). Within the Conversation Node we define an `npcGreetingMsg` (we could use an `npcGreetingList` instead) to display what Joe does and says to start the conversation. We can also use an `npcContinueMsg` (or `npcContinueList`) to contain Joe's further prompting if the player fails to respond with a conversational command (to create the impression that Joe does really want a reply). In this case, we'll have Joe pose a question that requires a simple yes or no answer, which we can deal with using a `YesTopic` and a `NoTopic` (rather than having to define any `SpecialTopics` or whatever). The new `ConvNode` and its associated topics then look like this:

```
+ ConvNode 'burner-spade'
  npcGreetingMsg = "<.p>He looks up at your approach, and walks
   away from the fire to meet you. <q>Have you finished with my spade
   yet?</q> he enquires anxiously.<.p>"
  npcContinueMsg = "<q>What about my spade? Have you finished with it
   yet?</q> {the burner/he} repeats anxiously. "
;

++ YesTopic
   "<q>Yes, I have.</q> you reply.<.p>
   <q>Can I have it back then, please?</q> he asks. "
;

++ NoTopic
   "<q>Not quite; can I borrow it a bit longer?</q> you ask.<.p>
   <q>Very well, then.</q> he conceded grudgingly, <q>But I need it
   to get on with my job, so please be quick about it.</q>"
;
```

The reason we start the `npcGreetingMsg` with the pronoun 'he' rather than `{The burner/he}` is that in the only context in which this message will ever be displayed, the player will just have read "Joe Black/The charcoal burner is walking round the fire, frowning as he keeps instinctively reaching for the spade that isn't there", so the burner's name doesn't need repeating immediately afterwards.

All that remains is to decide where to insert the call to `initiateConversation`. At first sight the obvious candidate would be in the `afterTravel(traveler,` `connector)` method of `burnerWaiting`, since this will be called after the Player Character travels to Joe's location:

```
  afterTravel(travler, connector)
  {
     getActor().initiateConversation(burnerFretting, 'burner-spade');
  }
```

Note the use of `getActor()` to get the Actor the current state belongs to. We could just as well have used `burner.initiateConversation` here, but there may be cases where `getActor` would be preferable (for example if one were defining a custom `TopicEntry` class for use in a number of different actors).

At this point it might be worth playing the game through to check that everything works properly and the game is still winnable. In the next chapter we'll add some more complications.

5. *Quick Summary*

This chapter has been mainly concerned with the implementation of further obstacles. In the course of this we have seen how to implement a lockable door, a see-through window, and ground you can dig in. We have expanded the range of ways of interacting with NPCs, and have shown how to make the contents of one location visible from another. One of the most important new concepts we have encountered is the creation of a new verb (command). In brief, the new features we have encountered in this chapter are:

New Classes

```
Door
Lockable
LockableWithKey (can be open with any key listed in its keyList
     property)
Key

RoomConnector  (connects rooms defined in its room1 and room2
    properties)
DistanceConnector [list of locations connected]
SenseConnector

MultiLoc (mix-in class; exists in all locations defined in its
    locationList property)
Wearable
```

For Use with NPCs

```
AskForTopic
setCurState(newActorState)
initiateConversation(state, 'node')
npcGreetingMsg/ npcGreetingList
npcContinueMsg/npcContinueList
YesTopic / NoTopic
```

Defining New Verbs

```
DefineTAction(MyVerb)
VerbRule(MyVerb)
```

Miscellaneous

```
askForIobj(Action)
remoteInitSpecialDesc(actor)
```

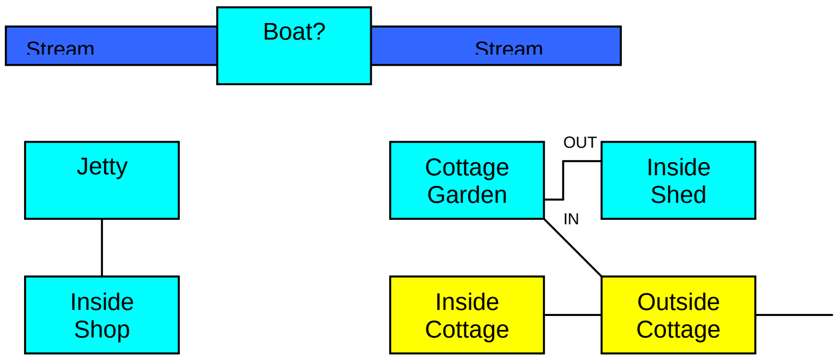
```
inRoomName(pov)
roomParts = [defaultFloor, defaultCeiling, defaultNorthWall etc]
gActionIs(Whatever)
isConversational
```

Chapter Seven - Pushing the Boat Out

1. *Let there be Light*

This is the last chapter in which we'll try to make things more difficult for poor Heidi. The complication we'll add is quite simple: simply change the class of `insideCave` from `Room` to `DarkRoom`. As you'll find if you now try to pay the cave a visit, Heidi now needs a light source to see what's going on there. The next task, then, is to plant a torch (which American readers may call a flashlight) somewhere. We can't put it inside the cottage, since that would make the game unwinnable (you need to dig up the boots to get to the key to get into the cottage). So instead we'll put the torch/flashlight in a garden shed. We'll also be creating a stream, a jetty, and a shop that Heidi will eventually need to visit in order to buy some batteries for the torch/flashlight.

Again, before seeing how this guide tackles all this, you might like to have a go at adding some of this for yourself. First of all, you need to add four more locations to the map: the garden, the inside of the shed, the jetty, and the shop, bearing in mind that the player may want to use the commands **enter shed** and **enter shop** as an alternative to other movement commands:



You'll also need to put a pair of oars and a torch (without batteries) inside the shed, perhaps placing the latter in a cupboard. There'll need to be an object representing the stream at both locations, and a customised response to **cross stream** explaining why Heidi can't simply cross it at that point. It would be especially neat if the player got the same response from a **north** command issued by the stream. You might also want to add some `FakeConnectors` or `NoTravelMessages` to explain why Heidi can't go west from the Garden or either east or west from the Jetty.

Your biggest challenge, however, will be to get Heidi from the Cottage Garden to the Jetty, since the map shows no direct connection. The idea is that Heidi gets there by rowing a boat down the stream (hence the oars), so you'll need a boat that Heidi can enter, a means of moving it between the Garden and the Jetty (and back), and a new **row** verb, which will require Heidi to be sitting in the boat holding the oars before it all works. Maybe the boat will need more than one object, such as an inside boat room that Heidi actually enters and a boat object to represent the boat from outside, just as you may create a shed object to represent the shed from outside.

If you can't manage all this by yourself, not to worry; once you've got as far as you can get, you can read on to see at least one way this can all be implemented.

First, then, we need to create the garden and its shed, using the opportunity to introduce a few more TADS 3 features we haven't come across yet:

```
cottageGarden : OutdoorRoom 'Cottage Garden'
  "This neat little garden is situated on the north side of the cottage. A
  stream runs along the bottom of the garden, while a short path disappears
  through a gap in the fence to the southeast, and another leads westwards
  down to the road. Next to the fence stands a small garden shed. "
  southeast = outsideCottage
  north : NoTravelMessage {"<<gardenStream.cannotCrossMsg>>"}
  east : NoTravelMessage {"You can't walk through the fence. "}
  west : FakeConnector {"That path leads down to the road, and you don't
    fancy going near all those nasty, smelly, noisy cars right now. " }
  in = insideShed
;

+ Decoration 'wooden fence' 'wooden fence'
  "The tall wooden fence runs along the eastern side of the garden, with
   a small gap at its southern end. "
;

+ gardenStream: Fixture 'stream' 'stream'
  "<<cannotCrossMsg>>"
  dobjFor(Cross)
  {
    verify() {}
    check() { failCheck(cannotCrossMsg); }
  }
  cannotCrossMsg = ' The stream is quite wide at
    this point, and too deep to cross. '
;

+ Enterable -> insideShed 'garden shed' 'garden shed'
  "It's a small, wooden shed. "
  matchNameCommon(origTokens, adjustedTokens)
  {
    if(adjustedTokens.indexOf('shed'))
      return self;
    else
      return cottageGarden;
  }
;

insideShed : Room 'Inside the Garden Shed'
 "The inside of the shed is full of garden implements, leaving just about
  enough room for one person to stand. An old cupboard stands
  in the corner. "
 out = cottageGarden
;

+ Decoration 'garden implements/hoe/rake/shears' 'garden implements'
  "There's a hoe, a rake, some shears, and several other bits and pieces. "
  isPlural = true
;

+ oars : Thing 'pair/oars' 'pair of oars'
  "The oars look like they're meant for a small rowing-boat. "
  bulk = 10
  initSpecialDesc = "A pair of oars leans against the wall. "
;
```

To take the simple points first, we add `isPlural = true` to the definition of the `Decoration` object so that an attempt to take, say, the hoe results in "The garden implements aren't important" rather than "The garden implements isn't important." The other simple point is that the `->insideShed` on the `Enterable` object is a shorthand way of specifying its `connector` property (through use of a template).

The more complex point involves the garden shed. Since it's called 'garden shed', the player could in principle refer to it either as 'shed', 'garden shed' or just 'garden', and all three forms would match. Yet one may feel that the last of these forms *shouldn't* match. The player character is standing in a garden, so logically the command **x garden** should result in a description of the garden, not the shed.

The `matchNameCommon` method is the way we get round this. To quote from the comments in the library source code:

> 'origTokens' is the list of the original input words making up the noun phrase, in canonical tokenizer format. Each element of this list is a sublist representing one token.
>
> 'adjustedTokens' is the "adjusted" token list, which provides more information on how the parser is analyzing the phrase but may not contain the exact original tokens of the command. In the adjusted list, the tokens are represented by pairs of values in the list: the first value of each pair is a string giving the adjusted token text, and the second value of the pair is a property ID giving the part of speech of the parser's interpretation of the phrase. For example, if the noun phrase is "red book", the list might look like ['red', &adjective, 'book', &noun].

For our purposes all we need to know is that `adjustedTokens` will be a list that will include all the tokens the player typed, so we can test whether or not 'shed' is among them. If not, the player must have typed 'garden' but not 'shed'. Since `adjustedTokens` is a list, we can use its `indexOf` method to find where in the list the string 'shed' is; if 'shed' is in the list then `adjustedTokens.indexOf('shed')` will return a non-zero number which the test (with if) will treat as true; if it isn't then the test will fail. If the test succeeds, the tokens include 'shed' and we return self (i.e. the shed) as the object matched. Otherwise the player typed 'garden' but not shed, so we return `cottageGarden` as the object matched. Thus, if the player types **x garden shed** or **x shed** the game will describe the shed, but if he or she types **x garden** it will describe the garden.

We could have implemented some of this functionality by using a *weak token* in the definition of the garden shed; we'd do this by enclosing the word 'garden' in parentheses in the list of vocabulary words, i.e.:

```
+ Enterable -> insideShed '(garden) shed' 'garden shed'
```

This would prevent the garden shed from responding to commands that just use the word 'garden' but would not remap such commands to the garden object. This, however, could easily be achieved by adding a `vocabWords` property to the definition of the cottage garden thus:-

```
    vocabWords = '(cottage) garden'
```

Note that once again we can use the weak token feature, so that the garden can be referred to by **x garden** or **x cottage garden** but not simply **x cottage** (you might want to add a cottage decoration object to respond to the latter). In practice one would probably use the weak tokens method rather than defining a custom `matchNameCommon` method to achieve the result desired here, but the detour through `matchNameCommon` has illustrated how to use it.

There's a couple more things we may want to do with the Enterable representing the outside of this garden shed. If the player types **open shed** or **look in shed**, the standard library responses may be not just unhelpful but potentially misleading (perhaps suggesting that the shed is only a decoration object):

>**open shed**
That is not something you can open.

>**look in shed**
There's nothing unusual in the garden shed.

We can solve the first problem simply by making the shed an `Openable` as well as an `Enterable`. The second is perhaps most easily solved by having **look in shed** treated as **enter shed**, on the grounds that someone wanting to find out what's in the shed would go inside it. A final problem is that if the player examines the shed it will be described as a 'small wooden shed', but that, as things stand, **x small wooden shed** will provoke the response, 'You see no small wooden shed here'; we need to add 'small' and 'wooden' to the vocabWords of this object. Our revised shed exterior thus becomes:

```
+ Openable, Enterable -> insideShed 'small wooden (garden) shed'
  'garden shed'
  "It's a small, wooden shed. "
  dobjFor(LookIn) asDobjFor(Enter)
;
```

You may have noticed that the description of the shed's interior includes mention of an old cupboard. What we want to do next is to put a tin on the cupboard and a torch inside it. On the face of it we can't do this, since an object can be either a container (something you can put things in) or a surface (something you put things on) but not both at the same time. We could get round this by laboriously making our cupboard out of separate objects, but fortunately the TADS 3 library has already done most of this work for us with a class called `ComplexContainer`. For the details of how `ComplexContainer` works, you can consult the *Library Reference Manual* and the *TADS 3 Tour Guide*, but there's no need to do so right now; the implementation of our cupboard using this class becomes quite straightforward:

```
+ cupboard: ComplexContainer, Heavy 'battered old wooden cupboard'
  'old cupboard'
  "The cupboard is a battered old wooden thing, with chipped blue and
   white paint. "
   subContainer : ComplexComponent, OpenableContainer { }
   subSurface : ComplexComponent, Surface { }
;
```

Basically, the `ComplexContainer` delegates putting-in and putting-on type behaviour to the anonymous nested objects defined in its `subContainer` and `subSurface` properties. These nested objects must be of class `ComplexComponent`, but you can then mix-in whatever classed you want (which, logically, will normally be something like `Container` and `Surface` respectively). The empty braces {} then contain the space where we'd define any properties or methods of these nested objects; but here we don't need to, since all the relevant behaviour has already been defined on their superclasses.

The next task is to put objects in and on the cupboard:

```
++ tin : OpenableContainer 'small square tin' 'small tin'
  "It's a small square tin with a lid. "
  subLocation = &subSurface
  bulkCapacity = 5
;

+++ battery : Thing 'small red battery' 'small red battery'
  "It's a small red battery, 1.5v, manufactured by ElectroLeax
  and made in the People's Republic of Erewhon. "
  bulk = 1
;

++ torch : Flashlight, OpenableContainer 'small blue torch/flashlight'
   'small blue torch'
  "It's just a small blue torch. "
  subLocation = &subContainer
  bulkCapacity = 1
;
```

The main thing to note here is the special syntax for specifying the initial location of objects inside a `ComplexContainer`. We can still use the + syntax to show that an object is on or in (or under or behind) a `ComplexContainer`, but we need to specify which subobject of the `ComplexContainer` the object is actually located in. To do this we use the special `subLocation` property which can be used *only* for initialization. If we subsquently wanted to move an object into a part of a `ComplexContainer` we'd need to do so with an explicit `moveInto`, e.g. `torch.moveInto(cupboard.subContainer)`. Note also that what we assign to `subLocation` must be a property *pointer* (a property name preceded by &).

We make the torch an `OpenableContainer` so that we can insert the battery. The behaviour of the torch requires a little thought. By default an object of the `Flashlight` class will provide light if it's switched on and will stop doing so if it's switched off. This is what we want, with the added complication that it should only be possible to turn the torch on if the battery is in it. A further complication is that if the player insists on removing the battery while the torch is on, it should at once go out again. Here's the definition of the torch with all that extra handling added:

```
++ torch : Flashlight, OpenableContainer 'small blue torch/flashlight'
'small blue torch'
  "It's just a small blue torch. "
  subLocation = &subContainer
  bulkCapacity = 1
  dobjFor(TurnOn)
  {
    check()
    {
      if(! battery.isIn(self))
      {
        "Nothing happens. ";
        exit;
      }
    }
  }
  iobjFor(PutIn)
  {
    check()
    {
      if(gDobj != battery)
      {
        "{The dobj/he} doesn't fit in the torch. ";
        exit;
      }
    }
    action()
    {
      inherited;
      makeOpen(nil);
      achieve.addToScoreOnce(1);
    }
  }
  notifyRemove(obj)
  {
    if(isOn)
    {
      "Removing the battery causes the torch to go out. ";
      makeOn(nil);
    }
  }
  achieve: Achievement
    { desc = "fitting the battery into the torch"  }
;
```

There's nothing very difficult here, but note that we take the opportunity to make sure that the battery is the only object that can be put in the torch;[35] we automatically close the torch after the battery is inserted to avoid getting the battery mentioned in response to an **inventory** command when we're carrying the torch. The most important thing to note is the use of the `notifyRemove` method to handle the battery being removed from the torch; we use this since we can't be sure which command a player might use to do this, e.g. **take battery** or **remove battery from torch**. The other thing we do is to award a point for inserting the battery into the torch for the first time only. To do this we define an `Achievement` object nested on the (custom) `achieve` property, and call its `addToScoreOnce(points)` method in our `iObjFor(PutIn) action` method. We do it this way since there is no freestanding `AddToScoreOnce` *function* we can call, and we need the `Achievement` object so that it can keep track of whether its been used to award points before.

At this point, we need to adjust the original location, first to indicate that there's a path round to the side of the cottage, and second to provide the relevant connection:

```
outsideCottage : OutdoorRoom 'In front of a cottage'
   "You stand just outside a cottage; the forest stretches east.
   A short path leads round the cottage to the northwest. " //add this
   east = forest
   in = cottageDoor
   west asExit(in)
   northwest = cottageGarden // add this
;
```

Once again, you can now recompile the program and test it all out to check that it still works.

[35] In a simpler case this could have been achieved using the RestrictedContainer class and setting its validContents property equal to [battery]; this would enable us to remove the check() routine, but would require us to add both Openable and RestrictedContainer to the class list of the torch.

2. *Row My Boat*

Leaving the battery so near the torch perhaps makes things a little too easy. For the final complication we'll oblige Heidi to go and buy a battery, and just to make things interesting the way to the shop will be by rowing a boat along the stream (now you know what the oars are for). Since we're going to row this boat, we once again need to define a new verb:

```
DefineTAction(Row);

VerbRule(Row)
  'row' singleDobj
  : RowAction
  verbPhrase = 'row/rowing (what)'
;

modify Thing
  dobjFor(Row)
  {
    preCond = [touchObj]
    verify() { illogical('{You/he} can\'t row {that dobj/him}'); }
  }
;
```

There is a `Vehicle` class (a subclass of `NestedRoom`), but this is not really what we want for our boat. Instead we'll use three different objects to define our boat; a `Heavy, Enterable` to represent the boat as seen from the outside, an `OutdoorRoom` to represent its interior, and an anonymous object placed inside the `OutdoorRoom` to be the object of the Row action. This is how we fit the three together:

```
boat : Heavy, Enterable -> insideBoat 'rowing boat' 'rowing boat'
  @cottageGarden
  "It's a small rowing boat. "
  specialDesc = "A small rowing boat floats on the stream,
      just by the bank. "
  useSpecialDesc { return true; }
  dobjFor(Board) asDobjFor(Enter)
;

insideBoat : OutdoorRoom
  name = ('In the boat (by '+ boat.location.name + ')')
  desc = "The boat is a plain wooden rowing dinghy with a single
   wooden seat. It is floating on the stream just by the
  <<boat.location.name>>. "
  out = (boat.location)
;

+ Fixture 'plain wooden rowing boat/dinghy' 'boat'
  "<<insideBoat.desc>>"
  dobjFor(Take)
  {
    verify() {illogical('{You/he} can\'t take the boat - {you\'re/he\'s} in
      it!'); }
  }

  dobjFor(Row)
  {
    verify() {}
    check()
    {
     if(!oars.isHeldBy(gActor))
     {
      "{You/he} need to be holding the oars before you can row this boat. ";
        exit;
     }
    }
    action()
    {
      "You row the boat along the stream and under a low bridge, finally
        arriving at ";
      if(boat.isIn(jetty))
      {
        "the bottom of the cottage garden.<.p> ";
        boat.moveInto(cottageGarden);
      }
      else
      {
        "the side of a small wooden jetty.<.p> ";
        boat.moveInto(jetty);
      }
      nestedAction(Look);
    }
  }
;
```

There is little here that is really new; we have simply fitted existing things together in a new way. Perhaps the most complex of these is the way we have defined the `name` property of `insideBoat`. We have taken advantage of the fact that a property can contain an expression (in parentheses) to build up a name that shows not only that the player character is the boat but where the boat is. We also use `<<boat.location.name>>` in the description of the boat's interior, so that this also reports not only what the boat looks like but where it is. Finally, we set the `out` property of `insideBoat` to `boat.location`, so that whenever we go out from `insideBoat` we end up wherever the boat object is. We can thus achieve the actual travel by moving the boat object around. Finally, we use the `specialDesc` property of the boat object to display a message that the boat is floating on the stream, and define the `useSpecialDesc` method always to return true so that `specialDesc` is always used.

The code for handling the Row command first checks that the actor is holding the oars. If so, then it checks which of two locations the boat is currently in and moves it to the other, displaying a suitable message to show the outcome, and then performing a nested Look action to show that we've arrived at a new location.

One could almost do away with the anonymous object contained within insideBoat, by defining `vocabWords = 'boat'` on `insideBoat` itself and moving the handling of for Row and Take to `insideBoat`. The main reason for not doing this is that one gets quite a bizarre message if one types the command **row** without a direct object and the parser helpfully selects the `insideBoat` object.

There's one other minor refinement you may want to include on this boat. If you get in the boat and then sit or lie down, you'll find that you're described as being in the boat sitting or lying on the ground. The way to fix this is to give the boat a more appropriate floor object:

```
boatBottom : Floor 'floor/bottom/(boat)' 'bottom of the boat'
;

insideBoat : OutdoorRoom
  name = ('In the boat (by '+ boat.location.name + ')')
  desc = "The boat is a plain wooden rowing boat with a single wooden seat.
  It is floating on the stream just by the <<boat.location.name>>. "
  out = (boat.location)
  roomParts = [boatBottom, defaultSky]
;
```

You may also want to add the small wooden seat referred to in the description of the inside of the boat, but this can be left as an exercise for the reader (or you can look at the source code to heidi.t that came with this Guide). Note that the way we have specified `boatBottom`'s `vocabWords` (`floor/bottom/(boat)`) will automatically match 'floor of boat' and 'bottom of boat' – but not just 'boat'); once again we don't need to do anything special to take care of the 'of' in phrases like these.

This boat is fairly simple since it moves between only two locations. If we wanted more possible locations we'd need a more complicated implementation of the Row verb – or perhaps define two versions of it, RowUpstream and RowDownstream. In principle, however, the approach taken here could be extended to all sorts of vehicles.

Talking of destinations, we have yet to define the destination the boat arrives at when it's rowed from the bottom of the garden (although you may already have made your own attempt). Here's this guide's suggestion:

```
jetty : OutdoorRoom 'Jetty'
  "This small wooden jetty stands on the bank of the stream. Upstream
  to the east you can see a road-bridge, and a path runs downstream
  along the bank to the west. Just to the south stands a small shop. "
  west : FakeConnector {"You could go wandering down the path but you don't
    feel you have much reason to. "}
  east : NoTravelMessage {"The path doesn't run under the bridge. "}
;

+ Distant 'bridge' 'bridge'
  "It's a small brick-built hump-backed bridge carrying the road over
  the stream. "
;

+ Fixture 'stream' 'stream'
  "The stream becomes quite wide at this point, almost reaching the
   proportions of a small river. To the east it flows under a bridge, and
   to the west it carries on through the village. "
  dobjFor(Cross)
  {
    preCond = [objVisible]
    verify() {}
    check()
     { failCheck ('The stream is far too wide and deep to cross here. '; }
  }
;
```

The one new feature introduced here is the `Distant` class, which may be used for objects that can be seen from a location but are too far away to interact with. This location isn't quite finished, since there's still no shop. We'll add that in the next section; in the meantime you can try the current version of the game out to make sure you can row your boat.

3. *Going Shopping*

The next task is to add the shop. The definition can go straight after the code listed above (so that the shop exterior is placed in the `jetty` room). If you haven't already tried defining your own shop interior, you could do so now, remembering to add a counter and maybe some items for sale (which could just be Decoration objects for now). You could also try adding a bell on the shop's counter, which Heidi can ring for service.

Here's our version

```
+ Enterable -> insideShop 'small shop/store' 'shop'
  "The small, timber-clad shop has an open door, above which is a sign
   reading GENERAL STORE"
;

insideShop : Room 'Inside Shop'
 "The interior of the shop is lined with shelves containing all sorts of
  items, including basic foodstuffs, sweets, snacks, stationery, batteries,
  soft drinks and tissues. Behind the counter is a door marked 'PRIVATE'. "
 out = jetty
 north asExit(out)
 south : OneWayRoomConnector
     {
       destination = backRoom
       canTravelerPass(traveler) { return traveler != gPlayerChar; }
       explainTravelBarrier(traveler)
        { "The counter bars your way to the door. "; }
     }
;

+ Decoration 'private door*doors' 'door'
  "The door marked 'PRIVATE' is on the far side of the counter, and there
  seems to be no way you can reach it. The other door out to the jetty is to
  the north. "
;

+ Fixture, Surface 'counter' 'counter'
  "The counter is about four feet long and eighteen inches wide. "
;

++ bell : Thing 'small brass bell' 'small brass bell'
  "The bell comprises an inverted hemisphere with a small brass knob
   protruding through the top. Attached to the bell is a small sign. "
  dobjFor(Ring)
  {
    verify() {}
    check() {}
    action() {"TING!";}
  }
;

+++ Component, Readable 'sign' 'sign'
 "The sign reads RING BELL FOR SERVICE. "
;

+++ Component 'knob/button' 'knob'
  "The knob protrudes through the top of the brass hemisphere of the bell. "
  dobjFor(Push) remapTo(Ring, bell)
;

backRoom: Room
  north = insideShop
;
```

Only a few things need any explanation here. The definition of `backRoom` is minimal because the Player Character will never visit it – the location exists solely as somewhere for the shopkeeper to be when she's not in the shop. We thus define the `OneWayRoomConnector` south from the shop interior so that the Player Character can't pass but the shopkeeper can. Although two doors are mentioned (or at least implied) by the room description, we supply a `Decoration` object to represent them; a fuller implementation isn't necessary. The essential items are the counter and the bell on the counter that the customer must ring to attract attention. This introduces a new class, the `Component` class, which, as its name suggests, treats objects of that class as components of the object that contains them. The sign is also of class `Readable`, which makes it a more likely target for a **read** command; it would also allow **read sign** to produce a different description if we had overridden the `readDesc` property on the object, but that would be rather fussy here. We allow the player to ring the bell either with **ring bell** or **push knob**, the latter command remapping to the former. Since **ring** is not a verb defined in the library, we need to define it, which we can do by copying the definition of Row and making the few necessary changes:

```
DefineTAction(Ring);

VerbRule(Ring)
  'ring' singleDobj
  : RingAction
  verbPhrase = 'ring/ring (what)'
;

modify Thing
  dobjFor(Ring)
  {
    preCond = [touchObj]
    verify() { illogical('{You/he} can\'t ring {that dobj/him}'); }
  }
;
```

If we were designing this game for real, we'd probably want to populate the shop with a few more decoration objects, e.g. for the shelves, the items on the shelves, and a cash register on the counter; we'll be adding some of these later, the rest can be left, yet again, as an exercise for the reader. Right now we need to attend to what happens when the bell is rung; obviously more than just displaying the string 'TING' is required; we need to summon the shopkeeper.

There are several ways this could be done; the way we shall use here probably isn't the simplest or the most elegant, it's simply one that lets us try out some features of the library we haven't met yet. In brief, we'll cause the ringing of the bell to trigger a `SoundEvent`. We'll then add a `SenseConnector` between the inside of the shop and the back room so that the `SoundEvent` can be detected by the shopkeeper even when she's in the back room, but we also need to make the shopkeeper a `SoundObserver` so she'll be receptive to the sound. We'll then have the sound trigger a daemon on the shopkeeper to make her walk into the shop one turn later (a fuse would have done just as well, so it doesn't much matter which we use here.)

This probably sounds rather complicated, if not downright incomprehensible, so let's take it one step at a time. First, we need to define the `SoundEvent`:

```
bellRing : SoundEvent
  triggerEvent(source)
  {
    "TING!<.p>";
    inherited(source);
  }
;
```

We have made the `SoundEvent` responsible for producing the "TING!" so we've had to override its `triggerEvent(source)` method, otherwise the definition of `bellRing` would have been even simpler. The call to `inherited(source)` within `triggerEvent(source)` is absolutely vital here, since it's the inherited method (i.e. the behaviour defined on the class) that does all the work of notifying interested parties that the sound event has just happened. The `source` parameter is the object from which the sound is supposed to emanate. This is the bell, whose `dobjFor(Ring)` now needs to its action method redefined thus:

```
        action()       {      bellRing.triggerEvent(self);     }
```

Where `self`, of course, refers to the bell object. The next task is to make sure that the bell ring can be heard in the back room as well as the shop. To do that we need to define a `SenseConnector` between the two:

```
SenseConnector, Intangible 'wall' 'wall'
  connectorMaterial = paper
  locationList = [backRoom, insideShop]
;
```

If everything works as it should, giving the `SenseConnector` the name 'wall' should be unnecessary, but if something works unexpectedly and the parser wants to refer to this object, it's as well that it should have a recognizable name so we can see what's happening. Since the sound does notionally travel through the wall, that's a sensible name to give it. On the other hand, the player does not need to interact with this object in any way, so we make it of class `Intangible` (as well as `SenseConnector`), so that it does not have any physical presence. The `connectorMaterial` defines the senses this `SenseConnector` will pass: `paper` is predefined to be transparent to sound and smell but opaque to sight and touch; in this case we don't care one way or the other about smell, and since it does what we want with the other three senses, this will do fine.

Now all we have to do is to define the shopkeeper. At this point we shan't program all her behaviour, just what's needed to get her to respond to the bell ring:

```
shopkeeper : SoundObserver, Person 'young shopkeeper/woman' 'young
shopkeeper'
   @backRoom
"The shopkeeper is a jolly woman with rosy cheeks and fluffy blonde curls. "
  isHer = true
  properName = 'Sally'
  notifySoundEvent(event, source, info)
  {
    if(event == bellRing && daemonID == nil && isIn(backRoom))
       daemonID = new Daemon(self, &daemon, 2);
    else if(isIn(insideShop) && event==bellRing)
     "<q>All right, all right, here I am!</q> says {the
        shopkeeper/she}.<.p>";
  }
  daemonID = nil
  daemon
  {
     moveIntoForTravel(insideShop);
    "{The shopkeeper/she} comes through the door and stands behind the
      counter. ";
    daemonID.removeEvent();
    daemonID = nil;
  }
  globalParamName = 'shopkeeper'
;
```

The first new feature to note here is the addition of `SoundObserver` to the shopkeeper's class list. This allows us to define the `notifySoundEvent` method, which will be triggered by the bell ring.[36] Since the bell ring is the only `soundEvent` in the game we hardly need to test for it, but to be on the safe side we do so anyway `(if event == bellRing)`. At the same time we check that the shopkeeper is still in the back room and that the daemon is not yet operative. We also check to see if the bell is rung while she's in the shop so she can simply respond with a suitable remark.

The complicated part is setting up the daemon. A new daemon is created with a call to `new Daemon(obj, prop, interval)`, where obj is the object it refers to, prop is the method on that object that is called each time the daemon is invoked, and interval is the number of turns between each invocation of the daemon. Here we define the daemon to run the `daemon` method (note that the parameter is supplied as `&daemon`) on self (the shopkeeper) every second turn (this means she won't come into the shop until the turn after the bell is rung). Since we want to be able to stop the daemon again we need to store a reference to the daemon, which we do in the property `daemonID` (note that we could have called the daemon method and the reference property anything we liked).

The daemon method first moves the shopkeeper into the shop and displays a suitable message to announce her arrival. We use `moveIntoForTravel` rather than `moveInto` to move the shopkeeper since with the latter the library code tries to find a path to move her through, and may well end up moving her through the `SenseConnector` with dire consequences (i.e. a runtime error); `moveIntoForTravel` avoids this problem. Once the shopkeeper has moved the daemon has done its work, so we get it to tidy up after itself, first by calling `daemonID.removeEvent()`, and finally by resetting `daemonID` back to nil so we can easily test for there no longer being an active daemon.

In this particular case we could have achieved the same effect slightly easier by using a fuse rather than a daemon. Instead of

```
daemonID = new Daemon(self, &daemon, 2);
```

We could have written

```
daemonID = new Fuse(self, &daemon, 1);
```

(Note the change in the number from 2 to 1 to produce the same effect of the shopkeeper moving on the next turn). The use of the fuse would have avoided the need for the line:

```
daemonID.removeEvent();
```

We should still need to keep track of whether we had an active fuse (using `daemonID`, which we might rename `fuseID` had we used a fuse) in order to make sure that a second ringing of the bell while the fuse was still active did not cause the creation of a second fuse.

Having reached this point, we can start expanding the definition of the shopkeeper using ActorStates and TopicEntries as with Joe the Charcoal Burner; you might like to try this out for yourself before reading this guide's version over the page.

[36] Actually, it would have worked just as well to define notifySoundEvent on the shopkeeper without bothering with the SoundObserver mix-in class, but it's as well to use SoundObserver for the sake of future compatibility; in some future version of the library this class may do more.

```
+ sallyTalking : InConversationState
   specialDesc = "{The shopkeeper/she} is standing behind the counter
    talking with you. "
   stateDesc = "She's standing behind the counter talking with you. "
   nextState = sallyWaiting
;

++ sallyWaiting : ConversationReadyState
  specialDesc = "{The shopkeeper/she} is standing behind the counter,
    checking the stock on the shelves. "
  stateDesc = "She's checking the stock on the shelves behind the counter. "
  isInitState = true
  takeTurn
  {
    if(!gPlayerChar.isIn(insideShop) && shopkeeper.isIn(insideShop))
      shopkeeper.moveIntoForTravel(backRoom);
    inherited;
  }
;

+++ HelloTopic
  "<q>Hello, <<getActor.isProperName ? properName : 'Mrs Shopkeeper'>>,</q>
      you say.<.p>
    <q>Hello, <<getActor.isProperName ? 'Heidi' : 'young lady'>>, what can
    I do for you?</q> asks {the shopkeeper/she}."
;

+++ ByeTopic
   "<q>'Bye, then!</q> you say.<.p>
    <q>Goodbye<<getActor.isProperName ? ', Heidi' : nil>>.
    See you again soon!</q> {the shopkeeper/she} beams in return. "
;

+++ ImpByeTopic
  "{The shopkeeper/she} turns away and starts checking the stock on the
     shelves.<.p>"
;
```

There is scarcely anything new here. Note the use of the double angle-bracket construction in the `HelloTopic` and `ByeTopic` to vary what's said according to whether Sally and Heidi have exchanged names yet, and the separate `ImpByeTopic` to decide what should be displayed when the conversation is ended; if the conversation ends because Heidi stops conversing or walks out of the shop, Sally simply goes back to work. Heidi will be considered to have stopped talking if she fails to address a conversational command to Sally for the number of turns in the `attentionSpan` of Sally's current `InCoversationState`. By default this is four; it can be made effectively infinite by setting `attentionSpan` to `nil`.

The `takeTurn`() method is called once every turn that this is Sally's current `ActorState`. Here we use it to check whether Heidi is still inside the shop; if she isn't, and Sally still is, then we send Sally back to her back room. It may occur to you that the `takeTurn` method is effectively a kind of daemon; to produce the effect of Sally coming into the shop the turn after the bell is rung, we could simply have added a few extra lines of code to this `takeTurn` method, perhaps in conjunction with a custom property. We could have dispensed with the whole mechanism of `SoundEvent` and `SenseConnector`, and simply have added a line of code in the `dobjFor(Ring)` method of the bell to change the value of the custom property which the additional code in the `takeTurn()` method could test for. But then we'd have lost the opportunity to look at sensory events, sense connectors, fuses and daemons. If you want to try to do it the simpler way, by all means experiment.

Since the shopkeeper has been summoned by the ringing of the bell, she is likely to initiate the conversation rather than waiting to be addressed by her customer. To handle this, add the following line after `daemonID = nil;` at the end of the shopkeeper's `daemon` method:

```
       initiateConversation(sallyTalking, 'sally-1');
```

And then add the definition of the appropriate conversation node; a good place for it would be between the definition of the shopkeeper and the definition of `sallyTalking`:

```
+ ConvNode 'sally-1'
  npcGreetingMsg = "<q>Right, what can I get you?</q> she asks. <.p>"
;
```

We don't need to put any topics under this conversation node; its only function is to display the `npcGreetingMsg`.[37] Any topics can then be handled by the `sallyTalking`:`InConversationState`. Let's start by adding a few now (put them after the definition of `sallyWaiting`):

```
++ AskTellTopic [shopkeeper, gPlayerChar]
  "<q>I'm Heidi. What's your name?</q> you ask.<.p>
  <q>Hello, Heidi; I'm <<shopkeeper.properName>>,</q> she smiles.
  <<shopkeeper.makeProper>>"
;

+++ AltTopic
 "<q>I'm feeling really <i>very</i> well today; how are you?</q> you
  ask.<.p>
 <q>I'm feeling very well too, thanks.</q> she tells you. "
 isActive = (shopkeeper.isProperName)
;

++ AskTellTopic @burner
  "<q>Do you know {the burner/him}, the old fellow who works in the
   forest?</q> you enquire innocently.<.p>
  <q>He's not <i>that</i> old,</q> she replies coyly. "
;

++ AskTellTopic @tWeather
  "<q>Lovely weather we're having, don't you think?</q> you remark.<.p>
  <q>Absolutely,</q> she agrees, <q>and with luck, it should stay fine
   tomorrow.</q>"
;

++ DefaultAskTellTopic, ShuffledEventList
  [
    '<q>What do you think about ' + gTopicText + '?</q> you ask.<.p>
    <q>Frankly, not a lot.</q> she replies. ',
    '<q>I think it\'s really interesting that...</q> you begin.<.p>
    <q>Oh yes, really interesting.</q> she agrees. ',
    'You make polite conversation about ' + gTopicText + ' and
    {the shopkeeper/she} makes polite conversation in return. '
  ]
;
```

[37] Which means we didn't really need to use initiateConversation here; the simpler alternative (simply displaying a message and changing Sally's ActorState) is left as an exercise for the reader.

Most of this should be fairly familiar. Note that placing a list in square brackets, as in the `[shopkeeper, gPlayerChar]` in the first `AskTellTopic` means that the topic can be triggered by any of the objects in the list; so this topic will work equally well for **ask shopkeeper about herself** or **tell shopkeeper about yourself**. Note also the use of string concatenation (joining strings together with the + operator) in the `DefaultAskTellTopic` to allow the use of a variable element (`gTopicText`) in an EventList. The other slight novelty (unless you already experimented with it at the end of the previous chapter) is the use of a `Topic` object to talk about the weather; since the weather is not a physical object defined anywhere in the game, we don't have a game object to match it to. To cope with this type of situation, where you want to be able to converse about things that are not game objects, there is a special `Topic` class. In this case all we need define is:

```
tWeather : Topic 'weather';
```

There's nothing magic about the 't' with which I started the object name here; that's just a convention I use to mark it as a `Topic` object as opposed to an ordinary game object. Note that, unlike game objects, `Topic` objects are assumed to be known by default, so that they are always available to **ask about** and **tell about** commands.[38] This can be changed by setting defining the `isKnown` property of the topic to nil when it is defined, e.g. if a player is to be informed about a gruesome murder during a conversation, but does not know of it when the game begins, one might define the murder topic object thus:

```
tMurder : Topic 'gruesome murder'
    isKnown = nil
;
```

When the player then learns of the murder at a later point one could use the `gSetKnown(tMurder)` macro to set `tMurder.isKnown = true`.

4. *Handling Cash Transactions*

a. *Providing Goods and Money*

Although we have created a shop and a shopkeeper, we have yet to program the actual purchase process. This will turn out to be one of the most complex tasks we have attempted so far; money is a surprisingly difficult thing to handle in IF. We shall first try an approach with a couple of buyable items and four coins. We shall then discuss how this might be expanded and simplified to cope with more general cases, without trying to add a more general case to our game.

What Heidi needs to buy from the shop is a battery. To make things a bit more interesting we'll assume she can also buy a bag of sweets (that's 'candy' for all you folks on the western side of the Atlantic). The first thing to do, then, is to remove the

[38] On the other hand the player can’t ask or tell about something the player character doesn’t know about, so that, for example, if you try to get Heidi to ask the shopkeeper about the charcoal burner before Heidi has come across it, you’ll get the DefaultAskTellTopic rather than the specific burner AskTellTopic.

battery from the tin (where we last left it) and to create a sweets/candy object. Remove the + sign from in front of the battery, move it after the contents of `insideShop`, and then define the bag of sweets:

```
battery : Thing 'small red battery' 'small red battery'
  "It's a small red battery, 1.5v, manufactured by ElectroLeax
  and made in the People's Republic of Erewhon. "
  bulk = 1
;

sweetBag : Dispenser 'bag of candy/sweets' 'bag of sweets'
  "A bag of sweets. "
  canReturnItem = true
  myItemClass = Sweet
;
```

We define the bag of sweets as a `Dispenser` since we expect it to contain individual items (i.e. sweets) which can be taken from the bag (and returned to it, since we have defined `canReturnItem = true)`. We set `myItemClass = Sweet` to define the type of object we expect the bag to hold. We must next define the `Sweet` class; the following code is more or less lifted straight from the TADS 3 sample game (sample.t) changing 'coin' to 'sweet' throughout and adding a few more customisations relevant to sweets, most notably making `Sweet` inherit from `Food` as well as `Dispensable`. While we're at it we'll adapt code from the sample game to make the sweets list neatly (e.g. "there are 9 sweets (3 red, 3 yellow and 3 green)" rather than "there is a red sweet, a red sweet, a red sweet, a yellow sweet etc."). For this we need a `ListGroupParen` and an `ItemizingCollectiveGroup` along with definitions of a `Sweet` class and subclasses to define collections of basically similar objects. Since this is something of a decorative distraction from our main objective here (Heidi doesn't need the sweets for the player to win the game), I shall simply present the adaptation from the sample.t code as an example, without pausing to discuss it in any depth; if you like, you can just skip it all for now.

```
class Sweet : Dispensable, Food
  desc = "It's a small, round, clear, <<sweetGroupBaseName>> boiled sweet. "
  vocabWords = 'sweet/candy*sweets'
  location = sweetBag
  listWith = [sweetGroup]
  sweetGroupBaseName = ''
  collectiveGroups = [sweetCollective]
  sweetGroupName = ('one ' + sweetGroupBaseName)
  countedSweetGroupName(cnt)
        { return spellIntBelow(cnt, 100) + ' ' + sweetGroupBaseName; }
  tasteDesc = "It tastes sweet and tangy. "
  dobjFor(Eat)
  {
    action()
    {
      "You pop <<theName>> into your mouth and suck it. It tastes nice
       but it doesn't last as long as you'd like.<.p>";
       inherited;
    }
  }
;

class RedSweet : Sweet 'red - ' 'red sweet'
    isEquivalent = true
  sweetGroupBaseName = 'red'
;
```

```
class GreenSweet : Sweet 'green - ' 'green sweet'
  isEquivalent = true
  sweetGroupBaseName = 'green'
;

class YellowSweet : Sweet 'yellow - ' 'yellow sweet'
  isEquivalent = true
  sweetGroupBaseName = 'yellow'
;

sweetGroup: ListGroupParen
    showGroupCountName(lst)
    {
        "<<spellIntBelowExt(lst.length(), 100, 0,
           DigitFormatGroupSep)>> sweets";
    }
    showGroupItem(lister, obj, options, pov, info)
        { say(obj.sweetGroupName); }
    showGroupItemCounted(lister, lst, options, pov, infoTab)
        { say(lst[1].countedSweetGroupName(lst.length())); }
;

sweetCollective: ItemizingCollectiveGroup 'candy*sweets' 'sweets'
;
```

Finally, we put some sweets in the bag simply by defining a number of anonymous objects of the appropriate type; note that the class definitions already locate the sweets in the bag so the code required to create the sweets is minimal:

```
RedSweet;
RedSweet;
RedSweet;
RedSweet;
GreenSweet;
GreenSweet;
GreenSweet;
YellowSweet;
YellowSweet;
```

Rather than getting bogged down in a description of how all this works (for which see the comments in sample.t), we'll regard it for now as simply an exercise in copying and adapting boilerplate code and get on with the business of setting up shop (indeed, for the main purpose of the exercise you could simply skip all the above and leave the bag of sweets as a `Thing` object, since Heidi's ability to eat, examine and taste the sweets plays no essential role in the game).

The two objects so far created, `battery` and `sweetBag`, are the two objects that will be handed to Heidi when as she completes her purchases. With only four pounds at her disposal, however, she is not going to buy up the shop's complete stock of these items. In other words, there should be sweets and batteries on display before and after the sale. On the other hand, it would be good if Heidi could not simply reach out and take them; placing them on shelves out of reach behind the counter and defining them to be of class `Distant` would achieve this object. But once they're in sight, they'll be the obvious objects for the parser to select in response to a command referring to batteries or sweets – including any command we use to indicate what Heidi is interested in buying. It would therefore be useful to define some custom properties on these items that can be used when we come to code the transactions. Add the following code so that the shelves are contained directly in the shop (e.g. by placing them directly after the definition of `+++ Component 'knob/button' 'knob'`):

```
+ Distant, Surface 'shelf*shelves' 'shelves'
  "The shelves with the most interesting goodies are behind the counter. "
  isPlural = true
;

++ batteries : Distant 'battery*batteries' 'batteries on shelf'
  "A variety of batteries sits on the shelf behind the counter. "
  isPlural = true
  salePrice = 3
  saleName = 'torch battery'
  saleItem = battery
;

++ sweets : Distant 'candy/sweets' 'sweets on shelf'
  "All sorts of tempting jars, bags, packets and boxes of sweets lurk
   temptingly on the shelves behind the counter. "
  isPlural = true
  salePrice = 1
  saleName = 'bag of sweets'
  saleItem = sweetBag
;
```

The `salePrice` property should be fairly self explanatory; `saleItem` contains the object that will actually be handed over to Heidi, while `saleName` is a name that will be used to describe this object in the course of the transaction. Finally, we need to put some money where Heidi will find it. Since we've taken the battery out of the tin and left it empty, let's put the cash in the tin:

```
++ tin : OpenableContainer 'small tin' 'small tin'
  "It's a small square tin with a lid. "
  subLocation = &subSurface
  bulkCapacity = 5
;

class Coin : Thing 'pound coin/pound*coins*pounds' 'pound coin'
  "It's gold in colour, has the Queen's head on one side and <q>One
   Pound</q> written on the reverse. The edge is inscribed with the words
   <q>DECUS ET TUTAMEN</q>"
   isEquivalent = true
;

+++  Coin;
+++  Coin;
+++  Coin;
+++  Coin;
```

Note that we can create the Coin class between the tin and the Coin objects and still use the + notation without any difficulty (the tin object was defined previously and is repeated here only for the sake of convenience). By the way if pound coins seem just too British to you, feel free to change them to dollar bills, euro notes or anything else; the principles will remain the same (though you'll need to be sure you make your changes consistently throughout what follows).

b. *Making the Sale*

What we now want to achieve is for Heidi to be able to ask for an item, be told the price, and receive the item she's asked for once she's handed over the correct money. We shall assume that once she's suggested one transaction, she can't start a second until she's completed the first. We shall also prevent her buying more than one of each item (she doesn't have enough money to buy a second battery, if she buys two bags of sweets she'll have insufficient funds left to buy the battery and the game will become unwinable, and in any case we only have one of each type of object to give her). Although we could create a separate transaction object to keep track of all this, we might as well use the shopkeeper object.

To make things a bit easier, we'll treat an **ask for** command directed to the shopkeeper as equivalent to **ask about** (on the assumption that if Heidi asks about a battery she wants to know about buying it, which comes to much the same thing as asking for it). We'll do this when we come to it by using the combined `AskAboutForTopic`. The next thing we have to reckon with is that if Heidi hands over more than one coin at a time (e.g. because the player types **give shopkeeper three pounds**), although this will count as one player *turn*, it will be treated as three iterations of the code handling the giving of a single coin to the shopkeeper. The problem here is that in this situation we don't want the shopkeeper to respond as each coin is handed over, but only after the complete number of coins specified in the player's command have been handed over. One way to handle this is via a fuse: the handing over of the first coin creates a new fuse; the handing over of subsequent coins merely keeps track of how many coins have been handed over. Once the specified number of coins has been handed over the player's turn is complete, and the fuse will fire – the code in the method called by the fuse can then handle the shopkeeper's response to the aggregate number of coins handed over (which might be too few, too many, or just right for the item asked for).

The code for starting the fuse will need to be on the `GiveTopic` that handles the giving of coins, but we'll code the method the fuse calls on the shopkeeper. We also need to add several custom properties to the shopkeeper object to keep track of the transaction. The code to be added to the shopkeeper is the following:

```
shopkeeper : Person, SoundObserver 'young shopkeeper/woman' 'young
shopkeeper'
…
cashReceived = 0
 price = 0
 saleObject = nil
 cashFuseID = nil
 cashFuse
 {
    if(saleObject == nil)
       {
         "<q>What's this for?</q> asks {the shopkeeper/she}, handing the
          money back, <q>Shouldn't you tell me what you want to buy
          first?</q>";
        cashReceived = 0;
       }
    else if(cashReceived < price)
      "<q>Er, that's not enough.</q> she points out, looking at you
        expectantly while she waits for the balance. ";
    else
    {
      "{The shopkeeper/she} takes the money and turns to take
       <<saleObject.aName>> off the shelf. She hands you
       <<saleObject.theName>> saying, <q>Here you are then";
      if(cashReceived > price)
        ", and here's your change";
      ".</q></p>";
      saleObject.moveInto(gPlayerChar);
      price = 0;
      cashReceived = 0;
      saleObject = nil;
    }
   cashFuseID = nil;
 }
;
```

The cashReceived property holds the number of coins that have been handed over to the shopkeeper in the current transaction; price is the number of coins needed in total to complete the transaction; saleObject is the object that will be handed over to the player on completion of the transaction; and cashFuseID points to the current fuse if there is one (we need this only so we can tell if there is a current fuse).

The cashFuse method is called when the fuse fires; if saleObject is nil Heidi has handed over some money without saying what she wants to buy with it, so we simply give the shopkeeper a suitable message to display, suggesting the player specifies what she or he wants to buy, and resetting cashReceived to zero ready for the next transaction. Otherwise, if a transaction is in process but the money handed over isn't enough to pay for the goods, the shopkeeper simply displays a message to the effect that she's expecting more cash. If however, there is a current transaction and enough money has been handed over, the routine moves the object requested (saleObject) to the player character, displays a suitable message, and resets all the relevant properties ready for a new transaction; if the player has actually handed over too much money an additional message is displayed to that effect. Finally, whatever else has happened, cashFuseID is reset to nil to show that there's no longer a current CashFuse.

The next job is to create the GiveShowTopic that will handle the handing over of coins. This will look a bit different from the GiveShowTopics we've seen before, both because of what it has to match, and because of what it has to do. We can't use the template because we have no way of specifying an object for this topic to match; instead is has to match any object belonging to the Coin class. We achieve this effect by overriding the matchTopic method; this method returns a score which is typically 100 for a good match and 0 for no match at all (the idea being that the TopicEntry with the highest score will be the one selected for matching); for any given TopicEntry the score is held in the matchScore property, so we make matchTopic return matchScore if an object is of class Coin and 0 otherwise.[39] This is also a good occasion for using the handleTopic method rather than the TopicResponse to handle the action, since it gives us access to the object that we want to manipulate, (as the obj parameter):

```
++ GiveShowTopic
   matchTopic(fromActor, obj)
   {
     return obj.ofKind(Coin) ? matchScore : 0;
   }
   handleTopic(fromActor, obj)
   {
     if(shopkeeper.cashFuseID == nil)
            shopkeeper.cashFuseID = new Fuse(shopkeeper, &cashFuse, 0);
     shopkeeper.cashReceived ++;
     if(shopkeeper.cashReceived > 1)
           "number <<shopkeeper.cashReceived>>";
     if(shopkeeper.cashReceived <= shopkeeper.price)
           obj.moveInto(shopkeeper);
   }
;
```

[39] For a fuller discussion of the matching score, see the article on 'Programming Conversations with NPCs' in the *Technical Manual*.

The `handleTopic` method will be called once for every coin that's handed over; for example, if the player types **give three pound coins to shopkeeper**, it will be run three times. We want a new Fuse created only the first time, so we first check whether `shopkeeper.cashFuseID` is `nil` before creating a new fuse and pointing the `shopkeeper.cashFuseID` property to it. We want to know how many coins are being handed over, so we increment `shopkeeper.cashReceived` each time through the loop. If the command involves multiple coins, the game will print "pound coin:" on a new line for each pass through the loop; to make this look slightly less superfluous we make it look like the coins are being counted out by printing "number two" etc. just after the "pound coin" display, but we don't do this the first time, in order to avoid an unnecessary "number one" if only one coin is handed over. Finally, we want to transfer the coins from the player character to the shopkeeper, but only up to the number of coins needed to meet the price asked for; any surplus coins are left in the player character's inventory. The whole `GiveShowTopic` should go in with the other topic entries under the `sallyTalking` state.

The final stage is to create the `AskAboutForTopic` objects (which will respond to either **ask for** or **ask about**) that will allow Heidi to request either a battery or a bag of sweets. The logic in each case is a little complicated, since there will be several things to check for (as we shall see). To avoid having to code this complicated logic twice over, we shall define a custom `BuyTopic` class (descended from `AskAboutForTopic`) which will handle all the complications, then simply create two `BuyTopic` objects, one for the battery and one for the sweets. Normally, one would use `AltTopic` to avoid burdening `TopicEntry` objects with a lots of `if… else…` type constructions, but since we want to encapsulate all the complexities of the behaviour in one class, we shall have to resort to `if` and `else` in the definition of that class:

```
class BuyTopic : AskAboutForTopic
  topicResponse
  {
    if(matchObj.saleItem.moved)
            alreadyBought();
    else if (shopkeeper.saleObject == matchObj.saleItem)
           "<q>Can I have the <<matchObj.saleName>>, please?</q> you ask.<.p>
            <q>I need another <<currencyString(shopkeeper.price -
             shopkeeper.cashReceived)>> from you.</q> she points out.<.p>";
    else if (shopkeeper.saleObject != nil)
          "<q>Oh, and I'd like a <<matchObj.saleName>> too, please.</q> you
            announce.<.p>
          <q>Shall we finish dealing with the <<shopkeeper.saleObject.name>>
            first?</q> {the shopkeeper/she} suggests. ";
   else
   {
        purchaseRequest();
        purchaseResponse();
        shopkeeper.price = matchObj.salePrice;
        shopkeeper.saleObject = matchObj.saleItem;
   }
  }
  alreadyBought = "You've already bought a <<matchObj.saleName>>.<.p>"
  purchaseRequest = "<q>I'd like a <<matchObj.saleName>> please,</q> you
    request.<.p>"
  purchaseResponse = "<q>Certainly, that'll be
    <<currencyString(matchObj.salePrice)>>,</q>
    {the shopkeeper/she} informs you.<.p>"
;
```

We provide the properties `alreadyBought`, `purchaseRequest` and `purchaseResponse` to allow easy customization of the messages displayed by a

BuyTopic, while at the same time providing acceptable default values for these properties that will allow a BuyTopic to be used without any customization. Note that we are using matchObj to get at the actual object that a given BuyTopic matches.

The topicResponse method then runs through a series of checks to trap the conditions under which we should not initiate a new transaction. First of all we check whether we've already purchased this object – note that the way we've things up the object purchased (moved into the player character's inventory) won't be the matchObj itself (which refers to the items sitting on the shelf) but the object referred to in the matchObj's saleItem property, so we check whether the latter has been moved; if it has, it's already been sold so we simply display the message defined in alreadyBought and take no further action.

The next condition we check for is whether the player is already part of the way through paying for the object requested. E.g. if the player typed **ask shopkeeper for battery** and then **give her two pounds**, there'd still be one pound to pay; here we trap the possibility that the player then types **ask shopkeeper for battery** again. If the transaction is already under way but incomplete, the saleObject property of the shopkeeper will have been set to the object asked for, so we test for this being the same as the saleItem corresponding to the matchObj. If it is, we display a message telling the player how much there is still to pay and take no further action.

The third possibility we have to eliminate is that the player may ask for one item, and then ask for another before the first transaction is complete; e.g. by entering the commands, **ask shopkeeper for battery**, **give her one pound**, **ask her for sweets**. If a transaction is in progress shopkeeper.saleObj will point to the object being purchased, since we have already tested for this being the object associated with matchObj, if we reach this point and shopkeeper.saleObj is not nil, it must be some other object. We accordingly display a message suggesting that the player should concentrate on buying one thing at a time.

Finally, if we have fallen at none of the preceding hurdles, we are in the position to set up a new transaction. This is fairly simple. First we display the player character's request (defined in purchaseRequest), which should normally say what Heidi wants to buy, then the shopkeeper's response (defined in purchaseResponse), which should say what the price is; the default values we define for these two properties will do this automatically, but these properties can be overridden to allow a greater variety of conversational interchanges at this point. Finally we set up the transaction by setting the two appropriate properties on the shopkeeper.

In a couple of places the code employs a custom function currencyString(amount), which simply returns a string spelling out an amount in pounds (e.g. currencyString(3) would return 'three pounds'). We can use the library function spellInt to do most of the work, so this function is defined simply as:

```
currencyString(amount)
{
  return spellInt(amount) + ' '  + ((amount>1) ? 'pounds' : 'pound');
}
```

If you are using dollars, euros, yen or denarii instead of pounds, remember to change this function accordingly.

Finally, we need to define the two BuyTopics to cope with the battery and the sweets. This then becomes very straightforward:

```
++ BuyTopic @batteries
   alreadyBought = "You only need one battery, and you've already bought
it.<.p>"
;

++ BuyTopic @sweets
   alreadyBought = "You've already bought one bag of sweets. Think of your
    figure! Think of your teeth!<.p>"
;
```

And this, apart from a few minor tweaks we shall be looking at in the next chapter, takes *The Further Adventures of Heidi* as far as this Guide is going to take them. If you compile and run the game again (after correcting any syntax errors and the other mysterious bugs that may have arisen through your mistyping or your computer's intrinsic cussedness), you should now be able to play it all the way through (which may take all of five minutes).

c. *Generalizing Financial Transactions*

The way we have defined `BuyTopic` would make it relatively easy to add to the items that Heidi could buy. All you would need to do is to define another object to sit on the shelf, a corresponding item to be handed over to Heidi, and the corresponding `BuyTopic`; to give a minimalist example:

```
/* Put this just after the shelf */
++ pears : Distant 'pear*pears' 'pears on shelf'
"A basket of fresh pears sits on the shelf behind the counter. "
    isPlural = true
    salePrice = 2
    saleName = 'pear'
    saleItem = pear
;

pear : Food 'pear' 'pear'
  "It's fresh-looking, green, and somewhat pear-shaped. "
 ;

/*Make sure this gets contained in sallyTalking */
BuyTopic @pears;
```

Provided you also add to the stock of pound coins (or whatever currency you're using) to cover the cost of all the items that could be purchased, it would be reasonably easy to going on adding as many buyable items as you wanted – provided they could all be priced in a small number of round pounds (or dollars, yen, roubles, drachmae, sesterces, euro, shekalim or whatever other currency takes your fancy). As soon as you want to start handling large amounts of money, and/or prices in pounds and pence (or dollars and cents etc.) the whole thing will start to become quite unwieldy. You certainly don't want to have to cope with handling individual twenty pound notes, ten pound notes, five pound notes, two pound coins, one pound coins, 50p, 20p, 10p, 5p, 2p and 1p coins in all possible combinations and permutations (dollars, dimes, nickels, quarters and cents would be quite bad enough). You'd do far better to define a single money object, with a value property stating how much money it represents at any one time, e.g.:

```
money : Thing 'cash/money' 'money'
  @outsideCottage
  "A quick count reveals that it comes to <<currencyString(value)>>. "
  value = 1204
  isPlural = true
;
```

Note that here we've chosen to store the value in the lowest denomination (pence, cents etc.) so any calculations can be handled as integer arithmetic (although you could always experiment with the `BigNumber` class as an alternative). One would then need to redefine the function `currencyString` to convert a value in pence, say, to a £12.04 display format.

```
function currencyString(amount)
{
   local valStr = ' &#163;';  /* £ sign; for dollars you could simply use
     '$' */
    valStr += (amount / 100);
    valStr += '.';
    local pence = amount % 100;
    if (pence < 10)
      valStr += '0';
    valStr += pence;
    return valStr;
}
```

The implementation of transactions would then become easier. They could be set up in exactly the same way (with a `BuyTopic`), but then one could implement a routine to respond simply to **give money to shopkeeper** or **pay shopkeeper**. This would simply have to check that enough money was available, and, if so, deduct it, e.g.

```
++ GiveTopic @money
topicResponse
{
    money.value -= shopkeeper.price;
    "You hand over the money and the shopkeeper gives you
        <<shopkeeper.saleObject.theName>>.<.p>";
     shopkeeper.saleObject.moveInto(gPlayerChar);
   if(money.value == 0)
  {
       "But you've used all your money!<.p>";
        money.moveInto(nil);
  }
}
;

+++ AltTopic
   "You don't have enough money to pay.<.p>"
   isActive = (shopkeeper.price > money.value)
;

+++ AltTopic
    "<q>What's this for?</q> asks {the shopkeeper/she}, handing the
     money back, <q>Shouldn't you tell me what you want to buy
     first?</q>"
    isActive = (shopkeeper.saleObject == nil)
;
```

Note that with this method there's no longer any need to use a fuse, so that all the `shopkeeper.cashFuse` method could be eliminated altogether. Similarly, since with this revised model there's no possibility of the player character issuing a command mid-transaction, so the definition of BuyTopic could be simplified considerably:

```
class BuyTopic : AskTopic
  topicResponse
  {
   if(matchObj.saleItem.moved)
        alreadyBought();
   else
   {
        purchaseRequest();
        purchaseResponse();
        shopkeeper.price = matchObj.salePrice;
        shopkeeper.saleObject = matchObj.saleItem;
   }
  }
  alreadyBought = "You've already bought a <<matchObj.saleName>>.<.p>"
  purchaseRequest = "<q>I'd like a <<matchObj.saleName>> please,</q> you
    request.<.p>"
  purchaseResponse = "<q>Certainly, that'll be
   <<currencyString(matchObj.salePrice)>>,</q>
  {the shopkeeper/she} informs you.<.p>"
;
```

In this way, the handling of the apparently more general and complex situation could actually be made rather simpler than the code we needed to handle four pound coins!

If you'd like to experiment with this, you could try it out in the Heidi game as an alternative to handling the four pound coins separately. Since most of the principles have now been spelt out, this may once again be left as an exercise for the reader.

5. *Quick Summary*

The main problem that has been exercising us in this chapter has been how to handle money. We have also introduced daemons and fuses, the implementation of a vehicle (a boat) and a torch (or flashlight), as well as a ComplexContainer which can be used to put things both in and on. In the course of seeing how it's all done we've encountered the following new library features:

New Classes:

```
AskForAboutTopic
ComplexContainer
Component
Dispensable
Dispenser
Distant
DarkRoom
Food
Heavy
Flashlight
OpenableContainer
SenseConnector (properties: connectorMaterial & locationList)
Topic

SoundEvent
SoundObserver (use with notifySoundEvent(event, source, info))

Daemon - new Daemon(obj, prop, interval)
Fuse - new Fuse(obj, prop, interval)
```

Other Stuff

```
attentionSpan
takeTurn() [method of ActorState]
isPlural
isEquivalent
bulk

matchNameCommon(origTokensm adjustedTokens)
Weak Tokens - designated by parentheses, e.g. '(garden) shed'

spellInt(number) [general purpose function]
addToScoreOnce(points) - method of Achievement class

'&#163' = '£' (pound sign) [example of HTML entity]
```

Chapter Eight - Finishing Off

1. *Filling in Some Gaps*

a. *Atmosphere Strings*

We have now taken the *Further Adventures of Heidi* about as far as it's worth taking them for the purposes of this Guide. It would be perfectly possible to go on adding in some further obstacles between Heidi and that ring (perhaps the oars could be hidden in a less obvious place, or the pound coins more widely dispersed), but it would become increasingly difficult to devise something that introduced a new feature of the library in a worthwhile way, and it might be better to leave such extensions as an exercise to any readers who wants to practice what they have learnt. Instead, we shall look at a few ways in which the library can help lend a bit more atmosphere to the game we've already created.

For the first example, consider the forest through which Heidi keeps passing. As it stands, the only other living creature she ever encounters there is Joe the charcoal burner; but one would expect a real forest to have all sorts of life in it. It's not worth creating lots of animal objects to represent the living forest, but we can use the `atmosphereStrings` property to simulate its presence. Rather than coding a separate `atmosphereStrings` for each room we consider to be part of the forest, it will be quicker and easier to define our own `ForestRoom` class that encapsulates this behaviour:

```
class ForestRoom : OutdoorRoom
   atmosphereList : ShuffledEventList
   {
     [
       'A fox dashes across your path.\n',
       'A clutch of rabbits dash back among the trees.\n',
       'A deer suddenly leaps out from the trees, then darts back off into
         the forest.\n',
       'There is a rustling in the undergrowth.\n',
       'There is a sudden flapping of wings as a pair of birds take flight
         off to the left.\n'
     ]
     eventPercent = 90
     eventReduceAfter = 6
     eventReduceTo = 50
   }
;
```

Defining a `ShuffledEventList` as a nested object should be familiar by now. The Room class has a built-in Room daemon that will call the `doScript` method of `atmosphereList`; in other words, all we have to worry about is defining the list for `eventList`, (a property defined in the template for the `EventList` class, so we don't need to name it explicitly), and they'll automatically be displayed. We can, however, exercise some control over the frequency with which they're displayed, and that's what the three properties `eventPercent`, `eventReduceAfter`, and `eventReduceTo` are for. As we have set it up, the messages we have defined will first be displayed 90% of the time, but after 6 turns this will fall to 50%. There is no need to define these properties at all; if none of them is overridden, then one of the messages in the list will be displayed each time the player character is in a room of class `ForestRoom`; if only the first is overridden, then that message frequency will be maintained throughout the game. The purpose of these properties is we can only define a finite number of strings, and after the player has seen them the first couple of times their appeal may start to wear a little thin; reducing their frequency may therefore help towards increasing the longevity of their positive contribution to the playing experience.

The three rooms that might best be redefined as belonging to the `ForestRoom` class are `forest`, `clearing`, and `forestPath`; you could also add define `fireClearing` to be of class `ForestRoom`, but in practice you'll probably find that the atmosphere strings tend to get in the way of the conversation with Joe.

b. *Sensory Emanations*

However, there is a different kind of atmospheric upgrade we could apply to the `fireClearing`. We have a fire billowing forth smoke, and so far we have smoke that makes its presence known to the nostrils only when the player explicitly chooses to smell it. In such a situation one might expect Heidi's nostrils to be assaulted by the smell of smoke whether she makes an active attempt to sniff it or not. We can simulate this by locating an `Odor` object directly inside our smoke object (having removed the `smellDesc` property from `smoke`), thus:

```
++ Odor 'acrid smoky smell/whiff/pong' 'smell of smoke'
  sourceDesc = "The smoke from the fire smells acrid and makes you cough. "
  descWithSource = "The smoke smells strongly of charred wood."
  hereWithSource = "You catch a whiff of the smoke from the fire. "
  displaySchedule = [2, 4, 6]
;
```

The `hereWithSource` message is displayed as part of the room description, and then at intervals defined in the `displaySchedule` list (in this case first after two turns, then after four turns, and finally every six turns; if we wanted to switch the smell messages off altogether we could end this list with `nil)`. This both stops the message from becoming too intrusive and repetitive, and also models the way in which human senses tend to take note of the environment. The property `sourceDesc` contains what will be displayed if you the player smells the object from which the smell emanates, in this case the smoke (i.e. it will be displayed in response to **smell smoke**), while the `descWithSource` property contains what will be displayed if the player refers directly to the `Odor` object, e.g. with **x smell** or **smell smoky whiff**. There are also `descWithoutSource` and `hereWithoutSource` properties that would contain messages to be displayed in the event of the source of the odour, in this case the smoke, being obscured from the player character; but since in this case the smoke from the fire is all too visible we don't need to define these properties (if, however, one had, for example, a dead rat concealed in a sandwich box, one could then use these two properties to contain messages appropriate to the situation before the player opens the box and discovers the source of the offensive odour; whether and where to include "You smell a rat" I leave to the discretion of the reader).

If you want to vary the message displayed according to the schedule, you can override `hereWithSource` (and/or `hereWithoutSource` if appropriate) with a method that checks the `displayCount` property (which is reset to 1 each time after the object comes within sensory range after it has left it (in this case, each time you return to the fire clearing after having been somewhere else). For example:

```
hereWithSource
{
   switch (displayCount)
   {
       case 1:
           "You catch a whiff of the smoke from the fire. ";
            break;
       case 2:
           "You catch another whiff of smoke from the fire. ";
            break;
      default:
           "You catch yet another whiff of smoke from that wretched fire. ";
    }
}
```

An alternative, if you were not concerned about restarting the list every time the object came into scope, would be to define the `hereWithSource` method to call the `doScript` method of an `EventList` object (which could be a nested object attached to the custom property of your `Odor`).

Note that you can also define a `Noise` object in much the same way as the `Odor` object is defined here (there are a couple of examples in sample.t). Perhaps, for example, the fire is making a crackling sound – once again the implementation can be left as an exercise for the interested reader (begin by defining `++Noise 'crackling sound/noise' 'crackling sound'` directly after the fire object, and then follow precisely the same format as used for the `Odor` object, making the obviously necessary adjustments to refer to sounds instead of smells).

c. *Settling the Score*

The maximum score you can obtain by winning the game is 7. Clearly we need to know how to adjust the maximum score; this is done by adding the following to the `gameMain` object, (which you may already have done if you copied the startup code from the start of chapter 3).

```
       maxScore = 7
```

The whole gameMain object for Heidi should look like this:

```
gameMain: GameMainDef
     initialPlayerChar = me
     showIntro()
     {
       "Welcome to the Further Adventures of Heidi!\b";
     }
     showGoodbye()
     {
       "<.p>Thanks for playing!\b";
     }
     maxScore = 7
;
```

The other routines do what they say: `showIntro()` shows the introductory text for the game (which you can make more elaborate if you wish), while `showGoodBye()` shows a terminating message when the game ends.

Of course, you may think that there should be more opportunities for gaining points. In that case you can add more `addToScore` calls at the appropriate places, being careful to ensure than they can only be called once (or use the `addToScoreOnce` method on new `Achievement` objects), and then adjust `gameMain.maxScore` accordingly.

d. *Destination Names*

If at the start of the game you type the command **east** followed by the command **exits** you'll see the response:

Obvious exits lead south; west, back to the in front of a cottage; and northeast.

This is less than ideal; "back to the in front of a cottage" is not exactly elegant. And you'd get much the same thing if you tried, for example, to go north from this location and the game helpfully tried to display the valid exits. What's happening, of course, is that the game is using the name we gave to the first room ("In Front of a Cottage"), converting it to lower case, and then displaying that as a description of where the path west leads back to from the forest. In this case, though, we'd prefer to see something like "back to the front of the cottage". Well, that's what TADS 3 provides the `destName` property for. In order to fix the problem with the exit listing, all we need to do is to add the line:

```
destName = 'the front of the cottage'
```

to the definition of `outsideCottage`. In fact, you may recall when we first introduced the Room template, it contained a convenient slot for `destName` (because the need to change `destName` from the default is quite common). So instead of adding the above line, we could simply change the start of the definition of `outsideCottage` to:

```
outsideCottage : OutdoorRoom 'In front of a cottage'
   'the front of the cottage'
   "You stand just outside a cottage; the forest stretches east.
   A short path leads round the cottage to the northwest. "
```

Clearly, this is not the only place in the *Further Adventures of Heidi* where we need to do this. Another example would be the `topOfTree` room:

```
topOfTree : OutdoorRoom 'At the top of the tree' 'the top of the tree'
          "You cling precariously to the trunk, next to a firm,
           narrow branch. "
```

The principle should (hopefully) now be clear enough, so I'll leave it as an exercise to the reader to check down the other rooms that need a `destName` adding (some of them will be fine as they are) and to add `destNames` as appropriate. You can then compare your efforts with those in the heidi.t file that should have come with this guide.

e. *Stopping Sally's Misbehaviour*

Sally is actually a pretty well-behaved shopkeeper most of the time, but there is a particular set of circumstances under which her behaviour – or at least the way it's reported – can become a little odd. If Heidi enters the shop, rings the bell, and leaves immediately (**ring bell** followed by **north**), the player will still see the message about the shopkeeper entering the shop, and the shopkeeper will still start a conversation, even though her customer is not actually there. Although the player may be unlikely to enter this sequence of commands, we really ought to try to anticipate *anything* the player might type, and so we do need to fix this bug.

The easiest way to go about it is to stop the daemon code actually executing unless Heidi is in the shop. We can do that by rewriting the shopkeeper's `daemon` method thus:

```
daemon
 {
   if(gPlayerChar.isIn(insideShop))
   {
     moveIntoForTravel(insideShop);
     "{The shopkeeper/she} comes through the door and
         stands behind the counter.<.p>";
     daemonID.removeEvent();
     daemonID = nil;
     initiateConversation(sallyTalking, 'sally-1');
   }
 }
```

This will work reasonably well; since the daemon has been set up to execute on the second turn, once the bell is rung, the daemon will keep checking every second turn to see whether Heidi is inside the shop, and if she is, will then move the shopkeeper into the shop and start the conversation. Thus if Heidi leaves the shop immediately after ringing the bell, the shopkeeper won't move or start talking until Heidi returns. It is theoretically possible that Heidi could keep missing the shopkeeper by entering the shop on the odd turns and leaving again immediately, but it's unlikely that a player who's interested in having Heidi meet the shopkeeper would actually keep making that sequence of moves.

Nonetheless, this solution is not quite ideal. Probably what ought to happen is that the shopkeeper should come into the shop on the second turn after the bell is rung regardless of whether Heidi remains in the shop or not, but her movement into the shop should only be reported if Heidi is in the shop to see it. We can achieve this quite straightforwardly by employing a `SenseDaemon` in place of the plain `Daemon`. To do this, find the shopkeeper's `notifySoundEvent` method and change the line:

```
daemonID = new Daemon(self, &daemon, 2);
```

to read:

```
daemonID = new SenseDaemon(self, &daemon, 2, self, sight);
```

A `SenseDaemon` is a special kind of `Daemon` that executes as normal except that it only displays anything if the player character can sense a particular object (the source) with a particular sense. The last two parameters of the call to `new SenseDaemon` are the *source* and the *sense* involved. In this case the source is the shopkeeper (which can be referred to here as `self` since we are setting up this `SenseDaemon` in a method of the shopkeeper object) and the sense is sight. The effect of this is (if we revert to the older definition of the `shopkeeper.daemon` method) is that the shopkeeper will move on the second turn after the bell is rung, but will only be reported as moving if Heidi is there to see it.

There's still one thing left to fix; although it's okay for Sally to come into the shop in response to the bell regardless of whether Heidi stays there to meet her or not, she should only start talking to Heidi if Heidi has indeed remained in the shop. We could use the test `if(gPlayerChar.isIn(insideShop))` as before, but we could also employ a different test. In this case, since what we're really testing is whether Sally *can* talk to Heidi, it would be reasonable to use her `canTalkTo` method. The relevant part of the revised shopkeeper code then becomes:

```
shopkeeper : Person, SoundObserver 'young shopkeeper/woman' 'young
shopkeeper'
 @backRoom
 "The shopkeeper is a jolly woman with rosy cheeks and
    fluffy blonde curls. "
 isHer = true
 properName = 'Sally'
 notifySoundEvent(event, source, info)
 {
   if(event == bellRing && daemonID == nil && isIn(backRoom))
      daemonID = new SenseDaemon(self, &daemon, 2, self, sight);

   else if(isIn(insideShop) && event == bellRing)
     "<q>All right, all right, here I am!</q> says
          {the shopkeeper/she}.<.p>";

 }
 daemonID = nil
 daemon
 {
    moveIntoForTravel(insideShop);
    "{The shopkeeper/she} comes through the door and
       stands behind the counter.<.p>";
    daemonID.removeEvent();
    daemonID = nil;
    if(canTalkTo(gPlayerChar))
      initiateConversation(sallyTalking, 'sally-1');
 }

… // continue as before
;
```

f. *Finishing the Boat*

There's just a couple of problems with our implementation of the boat, which we might like to fix now.

First of all, if the player enters the command **row the boat** when Heidi is in the garden, the game will reply with "You can't row that.", which is not entirely true. We need to provide a more appropriate response here. Again this is something you can probably work out how to do yourself by now, so have a go at it before reading on.

The second problem is a bit more subtle. Suppose that after issuing the command **row the boat** when Heidi is in the garden, the player types **enter it** followed by **row it**. The game will now respond with "The word 'it' doesn't refer to anything right now." You can probably work out why: the parser thinks that 'it' refers to the object we used to implement the *outside* of the boat, but once Heidi's entered the boat, that object is no longer in scope. We can cure this by using the `getFacets` property. This holds a list of other objects that we, the game author, consider to be facets of the same object, so that once we've referred to any of the objects representing the boat, the pronoun 'it' can refer to any of its facets currently in scope. So, for example, if we give the name `rowBoat` to the previously anonymous Fixture we placed in `insideBoat` to act as the target of a **row** command, we can now define `getFacets = [rowBoat]` on the `boat` object, and conversely `getFacets = [boat]` on the `rowBoat` object, and having referred to one, we can freely use 'it' to refer to the other.

The definition of the boat then becomes:

```
boat : Heavy, Enterable -> insideBoat 'rowing boat/dinghy' 'rowing boat'
  @cottageGarden
  "It's a small rowing boat. "
  specialDesc = "A small rowing boat floats on the stream,
      just by the bank. "
  useSpecialDesc { return true; }
  dobjFor(Board) asDobjFor(Enter)
  dobjFor(Row)
  {
   verify()
   {
    illogicalNow('You need to be aboard the boat before you can row it. ');
   }
  }
  getFacets = [rowBoat]
;
```

Notice the use of the illogicalNow() macro for handling **row boat** when Heidi is standing on the bank. It is illogical to try to row a boat when we're not in it, but it is not illogical to try to row a boat under all circumstances, so even under these circumstances **row boat** is less illogical than, say, **row sky** or **row stream**.

g. *Other Suggestions – including an MultiInstance*

There are also a several other things you could add that don't involve anything we have not already seen, including various `Decoration` (or, where appropriate, `Distant`) objects to deal with things mentioned in various room descriptions but not otherwise implemented, and various additional `NoTravelMessages` or `FakeConnectors` to deal more elegantly with the boundaries of our game world (e.g., the village mentioned in the description of the jetty location should perhaps be implemented as a `Distant` object in that room, and one or two `FakeConnectors` should be added to the `meadow` to explain why the only way Heidi can leave it is back across the stream). Since these involve nothing new, the reader can try them for him or herself.

But one new thing of this type does suggest itself, and that is putting some trees in the forest, since allowing the player to experience the following would be less than optimal:

Deep in the Forest

Through the deep foliage you glimpse a building to the west. A track leads to the northeast, and a path leads south.

There is a rustling in the undergrowth.

>**x trees**
The word "trees" is not necessary in this story.

If one is in a forest, one can reasonably expect to find trees, but rather than defining a different "trees" decoration object in all forest locations, we can simply use an `MultiInstance` to place our 'trees' object in every `ForestRoom`:

```
MultiInstance
   instanceObject : Decoration { 'pine tree*trees*pines' 'pine trees'
   "The forest is full of tall, fast-growing pines, although the
          occasional oak,
    beach and sycamore can occasionally be seen among them. "
   isPlural = true
   }
   initialLocationClass = ForestRoom
;
```

The slight complication here is that we have to define the `instanceObject` as a nested object within the `MultiInstance` object. We define it to be of class `Decoration` since the only interaction the player will have with the trees is to look at them. The effect of this code is that the game will create an instance of the 'pine trees' `Decoration` object in every location of the `ForestRoom` class. The `MultiInstance` class saves us the bother of having to do this by hand.

We could have implemented these trees as a `MultiLoc`, and in this particular game there would have been no functional difference. Strictly speaking, though, `MultiInstance` is the more correct class to use here. The main use for a `MultiLoc` is for a single physical object that exists in more than one location by virtue of being situated at the border of two or more rooms. For example, a large town square with a fountain at its centre might be implemented as four rooms, with the central fountain being a `MultiLoc` that appears in each. It is physically the same fountain whether it is viewed from, say, the northeast or the southeast corner of the square, and if the Player Character throws a coin into the fountain from the northeast corner of the square, he or she should then be able to retrieve it from the fountain even after moving to another part of the square, since it remains the same physical fountain. A `MultiLoc` may also be used for a `Distant` object, such as a far-off mountain range or the moon, that is visible from a number of different locations, since once again it is the same physical object that is being represented (provided it appears identical from all the locations in question).

But in this case, we are not trying to implement the same clump of trees visible from all parts of the forest, but the fact that there are trees, albeit numerically different trees, in all parts of the forest. Since all these trees are functionally identical (apart from the sycamore tree in the clearing that we have implemented separately) we can use `MultiInstance` as a short-cut to creating them all over the forest. Although in this game it makes no practical difference to the player whether we use a `MultiLoc` or a `MultiInstance`, in general it may. If, for example, Heidi were exploring the forest by night, then `MultiLoc` trees illuminated in one room would appear illuminated in all rooms (since they represent the same physical object). This would mean that if Heidi dropped her torch/flashlight at one spot in the forest and then moved to another part of the forest without any illumination, she'd be in a totally dark room but still be able to examine the trees, which is probably not what we'd want. Using `MultiInstance` ensures that we do note get this sort of unwanted behaviour.

You might think that a problem here would be that if the player types **examine tree** while Heidi is in the sycamore tree clearing, the parser will ask, "Which tree do you mean, the pine trees or the tree?". But in fact the library automatically takes care of this by giving a `Decoration` object a lower 'logical rank' than a normal object; that means that if two objects are in scope which might match the same vocabulary, one of them being a `Decoration` object and the other not, the other will be chosen in response to an **examine** command. So in this case when the player types **examine tree** the parser will assume that it is the sycamore tree that is meant, without troubling the player with a disambiguation request. For a fuller discussion of 'logical rank' see p. 74 above (and the section on 'Action Results' in the *Technical Manual*).

One more thing you might like to consider is letting the player know that you have defined a couple of custom verbs (although in this case it's arguably superfluous). You can do that by overriding the `customVerbs` property of `InstructionsAction`, thus:

```
modify InstructionsAction
  customVerbs = ['ROW THE BOAT', 'CROSS STREAM', 'RING THE BELL' ]
;
```

Then, when players type an **instructions** command**,** your custom verbs will be included in the list of game verbs the instructions text tells them about.

2. *Counting the Cash*

That the handling of cash could actually be simplified if one stops to think about implementing a more general solution shows that there's often more than one way to make a mousetrap in code, and that the first workable solution one comes up with isn't necessarily the best, the easiest or the most elegant. Even if we wanted to stick to having four coins in our game rather than a more abstract concept of money, we could have handled it better, and produced a better-looking output as a result. So for the sake of completeness we'll look at another way this could have been handed, although it is not exactly for the faint-hearted and introduces some techniques that are really rather advanced for a *Getting Started* guide; it may thus be this is something you'll want to skip on first reading.

The way we went about it before, using a fuse to sum up the result of handing over multiple coins in one turn, is perfectly workable, but the library does offer another way of doing it which, if not a great deal simpler, at least offers better control over what is displayed to the player. This is illustrated by the routine for handing coins to Bob in the sample game. We can adapt that code to our situation by redefining the shopkeeper's `GiveTopic` object for coins thus:

```
++ GiveShowTopic
   matchTopic(fromActor, obj)
   {
      return obj.ofKind(Coin) ? matchScore : 0;
   }
   handleTopic(fromActor, obj)
   {
     shopkeeper.cashReceived ++;
```

```
      currency = obj;
      if(shopkeeper.cashReceived <= shopkeeper.price)
           obj.moveInto(shopkeeper);
      /* add our special report */
      gTranscript.addReport(new GiveCoinReport(obj));

         /* register for collective handling at the end of the command */
      gAction.callAfterActionMain(self);

    }
     afterActionMain()
     {
         /*
          *   adjust the transcript by summarizing consecutive coin
          *   acceptance reports
          */
         gTranscript.summarizeAction(
             {x: x.ofKind(GiveCoinReport)},
             {vec: 'You hand over '
               + spellInt(vec.length())+' ' + currency.name+'s.\n' });
        if(shopkeeper.saleObject == nil)
        {
          "<q>What's this for?</q> asks {the shopkeeper/she}, handing the
           money back, <q>Shouldn't you tell me what you want to buy
            first?</q>";
         shopkeeper.cashReceived = 0;
        }
     else if(shopkeeper.cashReceived < shopkeeper.price)
       "<q>Er, that's not enough.</q> she points out, looking at you
         expectantly while she waits for the balance. ";
     else
     {
       "{The shopkeeper/she} takes the money and turns to take
        <<shopkeeper.saleObject.aName>>
       off the shelf. She hands you <<shopkeeper.saleObject.theName>> saying,
        <q>Here you are  then";
       if(shopkeeper.cashReceived > shopkeeper.price)
               ", and here's your change";
       ".</q></p>";
       shopkeeper.saleObject.moveInto(gPlayerChar);
       shopkeeper.price = 0;
       shopkeeper.cashReceived = 0;
       shopkeeper.saleObject = nil;
     }
    }
  currency = nil
;
```

The first thing you should notice about this is that we have effectively moved the code from the shopkeeper's `cashFuse` method into the new `afterActionMain()` method of the `GiveShowTopic`. This does mean that we now have to refer to all the properties involved as `shopkeeper.whatever` instead of just `whatever`, which makes it look a bit more complicated (this might be an argument for redefining these all as properties of the `GiveShowTopic`, but that would involve corresponding changes on the `BuyTopic` definition, so we shall not do it here). It also means that we can remove the `cashFuse` code from the shopkeeper object and that we no longer need to set up the fuse at all.

Clearly this is not the whole story; we have also replaced the entire fuse mechanism. In effect the call to `gAction.callAfterActionMain(self)` in `handleTopic` does a job analogous to the call to `shopkeeper.cashFuseID = new Fuse(shopkeeper, &cashFuse, 0)` that it replaces, in that it registers that once we have iterated over all the coins being given in this command, we want to handle the aggregate result of the transaction in the `afterActionMain()` method of `self`, i.e. the current object. Note that unlike the code to create a new fuse, there is no need to check that this has not been called on a previous iteration, since the registration will only be effective first time round. So far, there is not a great gain of simplicity compared with using the fuse to do the same job, but we are at least using a library mechanism designed to do the job we want, rather than trying to invent our own ad hoc mechanism, and this does allow all the code for handling the giving of coins to be put on the appropriate `GiveShowTopic`.

But we have not exhausted what this alternative way of designing this a particular mousetrap can do for us, even though the part that remains is frankly not the easiest thing to grasp first time round. The trouble with the way we did it before was that for each coin Heidi handed over to Sally the shopkeeper (if there were several), the game reported "pound coin: " on a new line. We mitigated this a little by trying to make it look as if the coins were being counted out:

```
pound coin:
pound coin: number 2
pound coin: number 3
```

But it really would have been better to do away with that repeating 'pound coin:' altogether (especially in a situation where you might want to hand over dozens of the things at a time), and simply to have one summary report that says something like "You hand over three pound coins." Well, this is what this new way of doing things allows us to do.

Firstly, we define what our own output for each line should be through the call to `gTranscript.addReport(new GiveCoinReport(obj))`. This actually does two things for us; first it allows us to define what will reported if only a single coin is handed over, and secondly it gives us a class name we have defined ourselves (`GiveCoinReport`) which we'll be able to use to manipulate the final report displayed if there's more than one `coin` handed over.

For this to work, we need to define the `GiveCoinReport` class:

```
class GiveCoinReport: MainCommandReport
    construct(obj)
    {
        /* remember the coin we accepted */
        coinObj = obj;

        /* inherit the default handling */
        gMessageParams(obj);
        inherited('You hand over {a obj/him}. ');
    }

    /* my coin object */
    coinObj = nil
;
```

The construct method – the object constructor – is called when we create a new object of the `GiveCoinReport` class through a call to `new GiveCoinReport(obj)`; the new object's `coinObj` property is set to the obj passed as a parameter, and, more interestingly for our purposes, we can customize the message that would be displayed each time a coin is handed over, but which will in fact only be displayed if a single coin is handed over in the turn. Here we customize it so it will read "You hand over a pound coin." (By using the parameter string `{a obj/him}` rather than the string literal 'pound coin' here we ensure that we'll still get a decent message if someone changes the coin name to 'dollar bill' or whatever).

Then comes the really clever (and complicated part); in order to replace the multiple reports of "You hand over a pound coin" that we'd otherwise see, we include the following code in the `afterActionMain()` method:

```
        gTranscript.summarizeAction(
              {x: x.ofKind(GiveCoinReport)},
              {vec: 'You hand over '
                 + spellInt(vec.length())+' ' + currency.name+'s.\n' });
```

This may well look Greek to you (unless you happen to know some Greek), but in brief we can at least say what it *does*: what it does is to remove every instance of the "You hand over a pound coin" report that would otherwise be displayed and instead prints the aggregate report "You hand over three pound coins." (or however many coins it was). We defined a custom `currency` property on the `GiveShowTopic` which is updated with the current object being handled every time `handleTopic` is invoked; that means that the currency property will refer to a coin object and we can use it to get at the name of the currency (rather than just assuming it's still called 'pound coin' after you've patriotically renamed it to dollar, euro or whatever). We form the plural by simply appending an 's', which will work as well for dollar bills as for pound coins; if, however, you've decided that your currency really has to be δραχμαί (yes, that's what Greek *really* looks like) for your game set in ancient Athens then you'd need to handle it a bit differently; perhaps by defining a `pluralName` property for your `Coin` class and using that instead of the `name` property here). The `spellInt(vec.length())` part of this string becomes a bit more manageable if one breaks it down step by step: the spellInt function takes an integer as an argument and returns the equivant spelt-out string (e.g. `spellInt(5)` returns 'five'). `vec` is going to be a vector (a kind of dynamically resizeable array) containing all the instances of the "You hand over a pound coin" message, so the length of the vector, i.e. the number of elements it contains, is equivalent to the number of coins handed over.

Even so, unless you're familiar with the code structure here, the line we're examining may *still* look rather like an arcane magical incantation; well, it's not *quite* that, but it's close to being the next best thing – a method call involving anonymous callback functions (if you don't feel any the wiser for being told that, don't worry; this is *not* the most self-evident topic). Rather than confuse you any further by trying to explain exactly what anonymous callback function are, I'll try to offer some explanation for what they do here. `gTranscript` is an object of the `CommandTranscript` class. We are invoking its `summarizeAction(cond, report)` method. But `cond` and `report` are not any old common-or-garden parameters of the sort we were all brought up or feel at least moderately comfortable with; it turns out that they are functions, functions that the `summarizeAction` method will use in the form `cond(cur)` and `report(vec)`. The first of these defines the condition that must apply to the report lines that we want to replace with our single summary report, and the second defines what that summary report will look like.

Our call to `gTranscript.SummarizeAction` is thus passing two arguments that are in effect short form function definitions. The first parameter, `{x: x.ofKind(GiveCoinReport)}`, in effect tells the `SummarizeAction` method to treat `cond()` as if it were defined as:

```
cond(x)
```

```
{
    return x.ofKind(GiveCoinReport);
}
```

You may remember that `GiveCoinReport` was the custom report class we defined a little way back, so what we're effectively telling the `SummarizeAction` with this is "look out for those reports of the `GiveCoinReport` class, they're the ones we want you to count up and replace for us."

Similarly, the second parameter is passed as `{vec: 'You hand over ' + spellInt(vec.length())+' ' + currency.name+'s.\n' }`. This effectively tells `SummarizeAction` to treat `report(vec)` as if it had been defined as:

```
report(vec)
{
    return 'You hand over ' + spellInt(vec.length())+' ' +
      currency.name+'s.\n';
}
```

Since the wizardry performed by `SummarizeAction` will have gathered up each instance of a `GiveCoinReport` into `vec`, when it uses this function to print the summary report, we'll get the result we want. If you don't understand all this at a first read-through, don't worry; reach for the nearest bottle of aspirins and read the description of anonymous functions and callbacks in the *System Manual*. If it still doesn't make too much sense to you first time round, you're doubtless in good company. But even if it takes you a little time to feel reasonably confident that you actually *understand* it, you may hopefully be able to *use* this example by treating the relevant code as piece of boilerplate in which you can slot in what you need for your own purposes; hopefully it'll soon become clear enough for you to see where you need to slot in what, even if the rest of it still seems less than intuitively obvious. In particular, what you need to do is to (a) define a `MyReport` class (substitute the name you actually use!); (b) supply the first argument to `gTranscript.summarizeTranscript` as `{x: x.ofKind(MyReport)}` and (c) supply the third argument as `{vec: 'My description of what happens to the ' + spellInt(vec.length())+' thingies that have been processed.\n' }`

3. *Looking Through the Window*

You'll recall that some time back we fitted a window to the cottage that allowed Heidi to see what's inside when she's outside and *vice versa*. As implemented, this is a rather 'passive' window that simply makes whatever's inside the cottage visible on the outside (and *vice versa*). Since a window is something one might actively look through, it would be nice if we could implement a **look through window** command, the response to which was a description of what was on the other side of the window. In principle, this could be done quite simply by something like this:

```
cottageWindow : SenseConnector, Fixture 'window' 'window'
  "The cottage window has a freshly painted green frame. The glass
    has been recently cleaned. "

  dobjFor(LookThrough)
  {
    verify() {}
    check() {}
    action()
    {
      local otherLocation;
      if(gActor.isIn(outsideCottage))
      {
         otherLocation = insideCottage;
         "You peer through the window into the neat little room
         inside the cottage. ";
      }
      else
      {
         otherLocation = outsideCottage;
         "Looking out through the window you see a path
          leading into the forest. ";
      }
      gActor.location.listRemoteContents(otherLocation);

    }
  }
  connectorMaterial = glass
  locationList = [outsideCottage, insideCottage]
;
```

The logic of this should be reasonably easy to follow: according to whether Heidi is inside or outside the cottage the `action` routine displays a brief general description of what can be seen on the other side of the window and then calls `listRemoteContents(otherLocation)` to list the contents of the other location being viewed through the window. We need to use a `listRemoteContents` routine because we want to list the contents of the other location as they appear from the room we're in, not as they would appear if Heidi were in the same location as they. This is all fairly straightforward apart from one thing: there is no `listRemoteContents` method in the library, so we'll have to provide our own.

The library does provide a routine that does almost what we want; it's called `lookAround`. This is normally used to provide a full room description, but we can restrict it to just listing the objects within a room. At a first approximation our listRemoteContents routine could be defined simply as:

```
listRemoteContents(otherLocation)
{
   lookAround(gActor, LookListSpecials | LookListPortables);
}
```

The first parameter of `lookAround` is the actor performing the command (i.e. `gActor`). The final parameter uses the bitwise or operator (|), the details of which are a bit beyond this Getting Started Guide; suffice to say that in this context it can be used to combine a number of option flags into a single argument. The two flags listed here are to list the objects with special descriptions and the portable objects.[40]

If you define `lookRemoteContents` on `Thing`, and then try compiling and running the game (with the revised `cottageWindow`), you'll find that it almost works as we want, but not quite. If you stand outside the cottage and look through its window on the first turn, the chair will be reported sitting in the corner of the room as expected. But if you subsequently collect the key, unlock the door, then drop the key

[40] The other available flags are LookRoomName, which would cause the the room name to be displayed, and LookRoomDesc, which would include the room description, neither of which we want in this case. For a standard verbose description, you can simply use the value `true` for this parameter, which is equivalent to LookRoomName | LookRoomDesc | LookListPortables | LookListSpecials.

on the ground, you'll find that looking in through the window from the outside lists the key as well as the chair, while looking out through the window from the inside lists the chair as well as the key. The problem is that `lookAround` lists everything that's visible from the room as well as the contents of the room itself, whereas we want something that lists only the contents of the room on the other side of the window.

Fortunately, TADS 3 provides another `Thing` method we can use to tweak this: `adjustLookAroundTable`. This is a method that can be used to remove any items we don't want included in the room description. By default it simply removes the point-of-view object, since an object looking round a location doesn't normally include itself in the list of things it sees; but we could use it to remove any object that's not in the location we're interested in:

```
adjustLookAroundTable(tab, pov, actor)
  {
    inherited(tab, pov, actor);
    if(listLocation_ !=  nil)
    {
      local lst = tab.keysToList();
      foreach(local cur in lst)
      {
        if(!cur.isIn(listLocation_))
          tab.removeElement(cur);
      }
    }
  }
```

In this method, we first call the inherited behaviour to remove the actor and point-of-view object. If `listLocation_` (which we'll explain more fully shortly) is not `nil`, we then go on to remove everything that's not in `listLocation_` from the table of items to be listed. The items that are otherwise about to be listed are in a `LookupTable` passed in the `tab` parameter. This `LookupTable` contains a series of pairs of values: a *key* containing the object and a value containing information about the sensing of the object. If that all sounds a bit too complicated, don't worry; all we're interested in this point is the list of objects contained in the keys. To get at this we use the `keysToList()` method. Then, having obtained the list of objects (in the local variable `lst`) we simply work through them and remove from the table every object that isn't in `listLocation_`.

So far, so good, but what exactly is `listLocation_` and how do we set it to what we want? Well, `listLocation_` is the name we're giving to the location (i.e. room) whose contents we want listed. It can't be passed as a parameter of `adjustLookAroundTable()`, since there's no provision for such an extra parameter in the library's definition of this method. So to make it available to `adjustLookAroundTable()` we need to define it as a property (which can then be set from another method). We add the underscore at the end of the name to highlight the fact that it's a property intended for a particular internal use only.

We next need to arrange for `listRemoteContents` to set `listLocation_` to the room whose contents we want listed:

```
listRemoteContents(otherLocation)
{
   listLocation_ = otherLocation;
   lookAround(gActor, LookListSpecials | LookListPortables);
   listLocation_ = nil;
}
```

Note that we reset `listLocation_` to `nil` after calling `lookAroundWithin`. This is vital, because we only want `adjustLookAroundTable` to remove items from the list of objects to be shown when `listRemoteContents` is called. At any other time we want `adjustLookAroundTable` to behave as defined in the library – which it will when `listLocation_` is `nil`. If `listLocation_` were not reset to `nil` at the end of each call to `listRemoteContents`, then subsequent listings of objects in room descriptions would cease to work properly (since all objects not in `listLocation_` would be removed from the list of objects to be shown). In fact, it's so important that we make sure that `listLocation_` is always reset to nil at the end of `listRemoteContents` that there's one more step we really ought to take to ensure that it always is – and that's to use `try… finally`. The way we do this is to enclose one or more statements in a block following the `try` keyword, then one or more statements in a block following the `finally` keyword. This will ensure that the statements in the `finally` block will *always* be executed, even if the game encounters an exception (such as an unanticipated error) in the `try` block. In this case we want to protect the call to `lookAroundWithin` in a `try` block and place the statement `listLocation_ = nil` in the `finally` block, to make it absolutely certain that `listLocation_` will *always* be reset to `nil` at the end of the method, come what may.

The modification to `Thing` required to achieve all this is thus:

```
modify Thing
  listLocation_ = nil
  listRemoteContents(otherLocation)
  {
    listLocation_ = otherLocation;
    try
    {
      lookAround(gActor, LookListSpecials | LookListPortables);
    }
    finally
    {
      listLocation_ = nil;
    }
  }

  adjustLookAroundTable(tab, pov, actor)
  {
    inherited(tab, pov, actor);
    if(listLocation_ !=  nil)
    {
      local lst = tab.keysToList();
      foreach(local cur in lst)
      {
        if(!cur.isIn(listLocation_))
          tab.removeElement(cur);
      }
    }
  }
;
```

If you don't totally understand all the details of this, don't worry at this stage. You can just copy the code and check that it works (or use it in your own project), and then come back to it when you're more experienced with TADS 3 and it makes more sense to you.

In the meantime, there's one further refinement we may want to add to the cottage window. At the moment, the chair inside the cottage advertises its presence as soon as Heidi's in `outsideCottage`, which she is on the very first turn of the game. It may be both a bit more subtle and a bit more realistic if she's not allowed to become aware of what's inside the cottage until she either enters it or explicitly looks in through the window.

The simplest way to achieve this is to have the window start totally opaque and only become transparent to sight the first time a **look through window** command is issued. To do this, first change the definition of `cottageWindow` so that instead of

```
connectorMaterial = glass
```

You have

```
connectorMaterial = adventium
```

Then add the statement

```
connectorMaterial = glass;
```

At the start of the `action` routine of `dobjFor(LookThrough),` say, immediately after `local otherLocation;`. Now, the chair will not be listed until the player issues the command **look through window**.

You might be tempted to add a further statement `connectorMaterial = adventium`; at the end of the `action` routine, so that the contents of the room inside the cottage are visible only when the window is being explicitly looked through. The problem with this is that once the player has seen the chair inside the cottage, he or she ought to be able to refer to it (e.g. with **x chair**), but making the window opaque again (adventium is a material that's opaque to all senses) will prevent that (**x chair** would result in the message 'you see no chair here', even though you'd just seen it through the window). In any case, once Heidi has once looked through the window it's not so unreasonable that she should continue to be aware of what lies on the other side of it.

4. *Easing Testing and Debugging*

A section headed 'testing and debugging' is always in danger of provoking a yawn from the reader of programming guides, so be assured that I shall be offering no general exhortations to good testing practice or complicated descriptions of debugging techniques. Instead I shall simply assume that you recognize at least some need to test and debug your creations and might be interested in one or tools that can make the process a bit less painful.

Even with a game as brief as *The Further Adventures of Heidi*, it can become quite tedious to have to keep retyping a whole sequence of commands to reach the point of the game at which you want to put something to the test (e.g., an alternative implementation of the way the chair object lets Heidi climb the tree). In a much larger game the prospect of having to do this would be simply horrendous. TADS 3 has a built-in mechanism for easing this pain: you can record a series of commands in a command (cmd) file and play them back on subsequent occasions. So, for example, if you wanted to test alternative chair implementations you might start up the game, and as the very first command type: **record**. A dialogue box will then appear asking you to supply a file name (you might call it 'chairtest'). You then carrying on issuing commands until to the point at which you want to make repeated tests, at which point you enter the command **record off**.

Then, on subsequent occasions, you can use the commands **replay**, **replay quiet** or **replay nonstop** to replay your command file to bring you back to the same point in the game. The first form of the command shows all the responses to each command as its read from the file, pausing to make you hit the space bar with every page-full of output; **replay nonstop**, as you might expect, does much the same thing, but without waiting for any keypresses (you can always scroll back the output window to read further back if you want to). Finally, **replay quiet** plays back the command file with no output to the screen at all; normally you'll want to issue a **look** command after a **replay quiet** command to check where it's brought you to. As with the **record** command, all three forms of the **replay** command provide you with a dialogue box to select the file you want to play back (although this can also be specified on the command line). There is also an analogous **script** which can be used to copy the entire output (player commands and game responses) to a log file; to stop outputting to the file you use the command **script off**. You might use this after making changes to a game to check that there were no unexpected changes to its transcript (perhaps by comparing before and after versions of the log file with a file comparison utility).

Although these are all helpful, it can also be useful (for testing purposes) to be able to teleport around the map or cause useful objects to teleport into the player character's hands from anywhere in the game world. Inform provides **gonear** and **purloin** verbs for just this purpose, but no such verbs exist in the TADS 3 library.[41] It is perfectly possible to implement your own versions, though; the main complication being that it is far from immediately obvious how to redefine the normal scoping rules to allow a command to refer to and act on an object that would normally not be considered in scope.

The quick and dirty way round this would be to override the `objInScope` method of the purloin and gonear actions:

```
DefineTAction(Gonear)
 objInScope(obj) { return true; }
;
```

This works perfectly well, but it's theoretically less than ideal; we don't actually want *every* object to be in scope for a **purloin** or **gonear** command, since it makes no sense to use these verbs with (say) Topics, ActorStates or TopicEntrys. A theoretically more rigorous approach, which we'll look at just to see how it's done, is to build our own list of objects we want considered in scope for these commands, and then use that:

```
#ifdef __DEBUG

/* The purpose of the everything object is to contain a list of all usable
game objects
   which can be used as a list of objects in scope for certain debugging
verb.
 Everything caches a list of all relevant objects the first time its lst
method is called. */

everything : object
  /* lst_ will contain the list of all relevant objects. We initialize it to
    nil to show that the list is yet to be cached */
```

[41] Although the library extension ncDebugActions.t, which you can download from the IF-Archive, defines equivalent actions.

```
  lst_ = nil

  /* The lst_ method checks whether the list of objects has been cached yet.
   If so, it simply returns it; if not, it calls initLst to build it first
   (and then returns it). */

  lst()
  {
    if (lst_ == nil)
      initLst();
    return lst_;
  }

  /* initLst loops through every game object and adds it to lst_, unless
   it's a Topic, which we don't want included even in this universal scope.
   */

  initLst()
  {
    lst_ = new Vector(50);
    local obj = firstObj();
     while (obj != nil)
     {
        if(obj.ofKind(Thing))
            lst_.append(obj);
        obj = nextObj(obj);
     }
     lst_ = lst_.toList();
  }
;
```

There should not be a great deal that requires explanation. We head the section with the preprocessor directive `#ifdef __DEBUG` (note the double underscore before DEBUG) to ensure that our debugging verbs are compiled only into the debugging version of the game we use for testing, not in the final release version. The `initList` method uses a vector rather than a list since this is slightly faster in execution; the routine converts `lst_` to a list right at the end. The built-in functions `firstObj()` and `nextObj()` are used to iterate through every object we have defined in the game, and we use a test to include only objects descended from Thing (i.e. programming objects that represent physical game objects). Since all the objects are defined in the game code there is no need to build this list more than once, so the code builds the list only the first time the `lst()` method is called; otherwise it simply returns the `lst_` previously constructed. A game that used dynamically created objects might have to use a slightly different approach.

Defining the purloin verb is then only slightly more complex than defining another new verb:

```
DefineTAction(Purloin)
  cacheScopeList()
     {
       scope_ = everything.lst();
     }
;


VerbRule(Purloin)
  ('purloin'|'pn') dobjList
  :PurloinAction
  verbPhrase = 'purloin/purloining (what)'
;

modify Thing
```

```
  dobjFor(Purloin)
  {
   verify()
   {
    if(isHeldBy(gActor)) illogicalNow('{You/he} {is} already holding it. ');
   }
   check() {}
   action
    {
      mainReport('{The/he dobj} pops into your hands.\n ');
      moveInto(gActor);
    }
  }
;

modify Fixture
  dobjFor(Purloin)
  {
    verify {illogical ('That is not something you can purloin - it is fixed
     in place.'); }
  }
;

modify Immovable
  dobjFor(Purloin)
  {
    check()
    {
      "You can't take {the/him dobj}. ";
      exit;
    }
  }
;
```

This definition assumes that we want to be able to **purloin** the kinds of things that you could normally expect to pick up and carry around, but not things that are fixed in place. If the behaviour you want is different from this, you can define `dobjFor(Purloin)` routines accordingly.

The definition for **gonear** is similar:

```
 DefineTAction(Gonear)
   cacheScopeList()
     {
       scope_ = everything.lst();
     }
;

VerbRule(Gonear)
  ('gonear'|'gn'|'go' 'near') singleDobj
  :GonearAction
  verbPhrase = 'gonear/going near (what)'
;

modify Thing
  dobjFor(Gonear)
  {
    verify() {}
    check() {}
    action()
    {
       local obj = getOutermostRoom();

       if(obj != nil)
       {
         "{You/he} {are} miraculously transported...</p>";
```

```
          replaceAction(TravelVia, obj);
        }
        else
          "{You/he} can't go there. ";
     }
   }
;

modify Decoration
  dobjFor(Gonear)
  {
    verify() {}
    check() {}
    action() {inherited;}
  }
;

modify Distant
  dobjFor(Gonear)
  {
    verify() {}
    check() {}
    action() {inherited;}
  }
;
```

What the gonear verb does is to transport the player character to the room in which the direct object of the gonear command is located (e.g. **gonear burner** would transport you the fire clearing). Using `getOutermostRoom` in the `action` method of `dobjFor(Gonear)` on `Thing` ensures that you are transported to the outermost container (the room), not the immediate container, which might be some other object. For example, if you enter the command **gonear torch** you'll end up inside the shed, not the cupboard (assuming the torch hasn't moved). If you added vocabulary words to particular rooms, you could also use the gonear verb with the room name to go straight to a room. We add definitions on `Decoration` and `Distant` since it makes perfectly good sense to gonear objects of these classes, but the library definition of these classes, which makes use of `dobjFor(Default)`, would otherwise annul the definition of `dobjFor(Gonear)` we put on `Thing`.

There may be other classes for which you'd want to add special handling for these verbs, but one in particular we need to consider is `MultiLoc`. Allowing a `MultiLoc` to be purloined might create havoc with your game world, while attempting to gonear a `MultiLoc` has no defined outcome; we thus need to define special handling to deal with these cases:

```
modify MultiLoc
  dobjFor(Gonear)
  {
    verify() { illogical('{You/he} cannot gonear {the dobj/him}, since it
        exists in more than one location. '); }
  }
  dobjFor(Purloin)
  {
    verify() { illogical('{You/he} cannot purloin {the dobj/him}, since it
       exists in more than one location. '); }
  }
;

#endif
```

We could simply have excluded MultiLocs from the scope list built by `everything.initLst()`, but this would result in slightly odd messages of the sort "You see no stream here" even when the stream is patently present in the location at which you issue an ill-advised **purloin stream** or **gonear stream** command. Allowing MultiLocs to be in scope and then providing a meaningful message explaining why the action is forbidden seems just that much neater. To be on the safe side you could add a similar modfication for MultiInstance (to trap **gonear trees** and **purloin trees**), but you'll find the game traps these for other reasons anyway.

The `#endif` preprocessor directive at the end balances the `#ifdef __DEBUG` at the start, thereby enclosing the entire block of code we've just defined to implement our two new testing and debugging verbs.

Note that have made this more complicated than strictly necessary; if you want to create this kind of thing for your own use you can dispense with the `everything` object and just define `objInScope(obj) { return true; }` on the `TAction` classes of your special debugging verbs; we have gone the longer route here to show how to build a custom scope list for cases where the blanket "put everything in scope" approach may not be what you want.

5. *Where to go from here*

We have only scratched the surface of the TADS 3 library (and the TADS 3 language), but hopefully enough has been covered here to get you started. The next step is for you to experiment on your own; you could either expand the *Further Adventures of Heidi* with further features of your own devising, or else start your own (and probably rather more interesting) test project. I shan't make any concrete suggestions what else to add to the Heidi game, since it'll be much more interesting for to implement your own ideas; but I'll pose a few questions your extension of the game might like to answer: whose cottage is it that Heidi starts by? Why were the boots buried in the cave? Could Sally the shopkeeper be the object of Joe's affections? Is the stream at the bottom of the garden the same as the stream Heidi has to cross to reach the meadow, and does that suggest any other uses for the boat? If Heidi could walk into the village from the jetty, what might she find there? Where else could she reach from the meadow? Why was the key lying there?

As an alternative, you might like to turn to the *Technical Manual* and read the article on Designing Your Game. This discusses the design of a game based at an airport, and shows how to implement the first few steps; as an exercise you could try implementing the rest of the game.

If you start following your own inventiveness, either by expanding the *Heidi* game or by writing your own, sooner or later (probably the former) you'll come up against something that hasn't been covered in this *Getting Started Guide*, or else come up against something that you can manage by some convoluted means but which makes you think, "surely TADS 3 provides a better way of going about it than this?" Given the richness both of the language and the library it's quite possible that it does, the problem is to discover where and how.

There are several other places you can look. One is the *TADS 3 Tour Guide* which, like the current *Getting Started* guide, takes you through the development of a complete game. The *Tour Guide*, however, is considerably longer, covers much more of the library, and tries to work through all the main classes in the library in a reasonably systematic fashion. It should prove an excellent resource for extending and deepening your knowledge of TADS 3 (but then I would say, since I wrote it!). The *TADS 3 Tour Guide* should have come with the documentation included with the TADS Author's Kit, but it can be found on line at (and various formats downloaded from) http://users.ox.ac.uk/~manc0049/TADSGuide/intro.htm. It is also available from the IF-Archive.

Another place to look is is the TADS 3 *Technical Manual*, which again should be included with the documentation that comes with the TADS 3 author's kit, or should otherwise be downloadable from http://www.tads.org. This contains a variety of sections, some of which cover in greater detail material introduced in this guide, and some of which which take you into new areas (such as writing a TADS 3 game in the past tense). A third resource is the TADS 3 *System Manual*, also included with the standard documentation (or downloadable from http://www.tads.org). This may not be something you want to read from cover to cover in one sitting, but it is something you will want to refer to again and again, and you will need to familiarize yourself with much of its contents sooner or later. The thing to do first time round is probably to try to read through all the bits that look interesting and informative, and to skip anything that seems boring and obscure unless and until you really need it.

Yet another resource is the 'TADS 3 Version History' (the link near the bottom of the HTML page when you select the Help option from Workbench), and in particular the Recent Library Changes list, which can be a mine of information on TADS 3 features.

How you tackle all this documentation is up to you, but here's one suggestion for an order that you might find helpful. First go to the *Technical Manual* and read the articles 'Tips on Designing Your Game', 'Object-Oriented Programming Overview', 'How to Create Verbs', 'Action Results', 'Message Substitution Parameters', 'NPC Travel' and 'Creating Dynamic Characters'. Next turn to the *System Manual* and read everything in Parts I-III up to and including 'Anonymous Functions'. Then read Part IV, but skip over 'Input Scripts', 'ByteArray', 'Character Set', 'File', 'Grammar Prod', 'Intrinsic Class' and 'StringComparator' for now. Then read the section on 'Program Initialization' from Part V After reading 'tads-io Function Set' read the article on 'Some Common Input/Output Issues' from the *Technical Manual*. After all that you may want to have a go at the *TADS 3 Tour Guide*, and you should be well equipped to benefit from it.

But no set of documentation is going to tell you everything you will eventually need to know about the TADS 3 library. Sooner or later there will be no subsitute for looking at the library code to see how various classes and functions are implemented. Fortunately, the library code is well commented to guide you through understanding how things work. Even more fortunately, the documentation that comes with the TADS 3 author's kit comes with a set of linked HTML files (The *TADS 3 Library Reference Manual*) that make finding your way round the library source files *much* easier than it otherwise would be. You will find this an invaluable reference resource whenever you're programming in TADS 3.

If you've followed this *Getting Started Guide* through to this point, you will not yet have created the most exciting work of Interactive Fiction in the known universe, but hopefully you will have been helped over the initial hump to start getting to grips with TADS 3. In any case, gaining mastery of TADS 3 isn't a matter of commiting every feature of the TADS 3 language and library to memory (few people could do that), but of learning enough to be able to carry out common tasks with ease, and to know where to look to find out how to do the rest with the minimum of difficulty. This *Getting Started Guide* should have set you well on the way down this path, and the remaining items in this set of documentation should carry you the rest of the way. From now on, the rest is up to you.

APPENDIX A – Action Message Properties

d = dObjFor i = iObjFor   A=Action C=Check   V=Verify

AskAbout	Thing dV	notAddressableMsg
AskFor	Thing dV	notAddressableMsg
AttachTo	Thing dV	cannotAttachMsg
	Thing iV	cannotAttachToMsg
Attack	Thing dA	uselessToAttackMsg
AttackWith	Thing dA	uselessToAttackMsg
	Thing iV	notAWeaponMsg
Board	Thing dV	cannotBoardMsg
Break	Thing dV	shouldNotBreakMsg
BurnWith	Thing dV	cannotBurnMsg
	Thing iV	cannotBurnWithMsg
Clean	Thing dV	cannotCleanMsg
CleanWith	Thing dV	cannotCleanMsg
	Thing iV	cannotCleanWithMsg
Climb	Thing dV	cannotClimbMsg
ClimbDown	Thing dV	cannotClimbMsg
ClimbUp	Thing dV	cannotClimbMsg
Close	Thing dV	cannotCloseMsg
Consult	Thing dV	cannotConsultMsg
ConsultAbout	Thing dV	cannotConsultMsg
CutWith	Thing dA	cutNoEffectMsg
	Thing iV	cannotCutWithMsg
Default	Decoration diV	notImportantMsg
	Distant diV	tooDistantMsg(self)
	Intangible diV	notWithIntangibleMsg
	Unthing diV	notHereMsg
Detach	Thing dV	cannotDetachMsg
DetachFrom	Thing dV	cannotDetachMsg
	Thing iV	cannotDetachFromMsg
DigWith	Thing dV	cannotDigMsg
	Thing iV	cannotDigWithMsg
Drink	Thing dV	cannotDrinkMsg
Drop	Immovable dA	cannotMoveMsg
Doff	Thing dV	notDoffableMsg
Eat	Thing dV	cannotEatMsg
Enter	Thing dV	cannotEnterMsg
EnterOn	Thing dV	cannotEnterOnMsg
Extinguish	Thing dV	cannotExtinguishMsg
Fasten	Thing dV	cannotFastenMsg
FastenTo	Thing dV	cannotFastenMsg
	Thing iV	cannotFastenToMsg
Flip	Thing dV	cannotFlipMsg
Feel	Thing dA	feelDesc ""
Follow	Thing dV	notFollowableMsg
GetOffOf	Thing dV	cannotGetOffOfMsg
GetOutOf	Thing dV	cannotUnboardMsg
GoThrough	Thing dV	cannotGoThroughMsg

GiveTo	Thing iV	notInterestedMsg
JumpOff	Thing dV	cannotJumpOffMsg
JumpOver	Thing dV	cannotJumpOverMsg
Kiss	Thing dV	cannotKissMsg
	Actor dA	cannotKissActorMsg
LieOn	Thing dV	cannotLieOnMsg
Listen	Thing dA	(soundDesc "")
Light	Thing	asDobjFor(Burn)
Lock	Thing dV	cannotLockMsg
	IndirectLockable dC	cannotLockMsg
LockWith	Thing dV	cannotLockMsg
	Thing iV	cannotLockWithMsg
	Lockable dV	noKeyNeededMsg
	IndirectLockable dC	cannotLockMsg
LookBehind	Thing dA	nothingBehindMsg
LookIn	Thing dA	lookInDesc ""
LookThrough	Thing dA	nothingThroughMsg
LookUnder	Thing dA	nothingUnderMsg
Move	Thing dA	moveNoEffectMsg
	Fixture dV	cannotMoveMsg
	Immovable dA	cannotMoveMsg
MoveTo	Thing dA	moveToNoEffectMsg
	Fixture dV	cannotMoveMsg
	Immovable dC	cannotMoveMsg
MoveWith	Thing dA	moveNoEffectMsg
	Thing iV	cannotMoveWithMsg
	Immovable dC	cannotMoveMsg
	Fixture dV	cannotMoveMsg
Open	Thing dV	cannotOpenMsg
PlugIn	Thing dV	cannotPlugInMsg
PlugInto	Thing dV	cannotPlugInMsg
	Thing iV	cannotPlugInToMsg
Pour	Thing dV	cannotPourMsg
PourInto	Thing dV	cannotPourMsg
	Thing iV	cannotPourIntoMsg
PourOnto	Thing dV	cannotPourMsg
	Thing iV	cannotPourOntoMsg
Pull	Thing dA	pullNoEffectMsg
	Fixture dV	cannotMoveMsg
	Immovable dA	cannotMoveMsg
Push	Thing dA	pushNoEffectMsg
	Fixture dV	cannotMoveMsg
	Immovable dA	cannotMoveMsg
PushTravel	Thing dV	cannotPushTravelMsg
	Fixture dV	cannotMoveMsg
	Immovable dA	cannotMoveMsg
PutBehind	Thing iV	cannotPutBehindMsg
	Fixture dV	cannotPutMsg
	Component dV	cannotPutComponentMsg(location)
	Immovable dC	cannotPutMsg
PutIn	Thing iV	notAContainerMsg
	Fixture dV	cannotPutMsg
	Component dV	cannotPutComponentMsg(location)
	Immovable dC	cannotPutMsg
PutOn	Thing iV	notASurfaceMsg

	Fixture dV	cannotPutMsg
	Component dV	cannotPutComponentMsg(location)
	Immovable dC	cannotPutMsg
PutUnder	Thing iV	cannotPutUnderMsg
	Fixture dV	cannotPutMsg
	Component dV	cannotPutComponentMsg(location)
	Immovable dC	cannotPutMsg
Screw	Thing dV	cannotScrewMsg
ScrewWith	Thing dV	cannotScrewMsg
	Thing iV	cannotScrewWithMsg
Search	Thing dA	as LookIn
SitOn	Thing dV	cannotSitOnMsg
ShowTo	Thing iV	notInterestedMsg
Smell	Thing dA	(smellDesc "")
StandOn	Thing dV	cannotStandOnMsg
Switch	Thing dV	cannotSwitchMsg
Take	Fixture dV	cannotTakeMsg
	Component dV	cannotTakeComponentMsg(location)
	Immovable dA	cannotTakeMsg
TakeFrom	Fixture dV	cannotTakeMsg
	Component dV	cannotTakeComponentMsg(location)
	Immovable dC	cannotTakeMsg
TellAbout	Thing dV	notAddressableMsg
TalkTo	Thing dV	notAddressableMsg
Taste	Thing dA	tasteDesc ""
Turn	Thing dV	cannotTurnMsg
	Immovable dA	cannotMoveMsg
TurnOff	Thing dV	cannotTurnOffMsg
TurnOn	Thing dV	cannotTurnOnMsg
TurnTo	Thing dV	cannotTurnMsg
TurnWith	Thing dV	cannotTurnMsg
	Thing iV	cannotTurnWithMsg
ThrowDir	Thing dA	dontThrowDirMsg *or* ShouldNotThrowAtFloorMsg
ThrowTo	Thing dV	willNotCatchMsg(self)
TypeLiteralOn	Thing dV	cannotTypeOnMsg
Unfasten	Thing dV	cannotUnfastenMsg
	Thing iV	cannotUnfastenFromMsg
UnPlug	Thing dV	cannotUnplugMsg
UnPlugFrom	Thing dV	cannotUnplugMsg
	Thing iV	cannotUnplugFromMsg
Unlock	Thing dV	cannotUnlockMsg
	IndirectLockable dC	cannotUnlockMsg
UnlockWith	Thing dV	cannotUnlockMsg
	Thing iV	cannotUnlockWithMsg
	Lockable dV	noKeyNeededMsg
	IndirectLockable dC	cannotUnlockMsg
Unscrew	Thing dV	cannotUnscrewMsg
UnscrewWith	Thing dV	cannotUnscrewMsg
	Thing iV	cannotUnscrewMsg
Wear	Thing dV	notWearableMsg

INDEX