QDK Tutorial

How to write games for

version 4.1

1 Introduction

1.1 What is Quest?

Quest is a system for text-based games, often referred to as **text adventures** or **interactive fiction**. It can also be used for delivering **computer-based training**, and has a variety of applications in **educational software**.

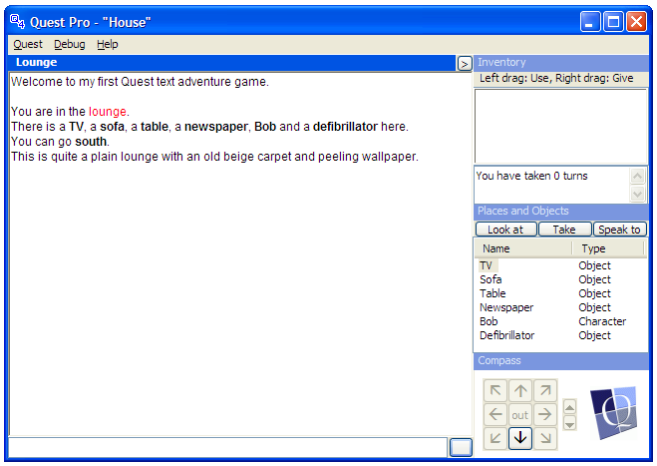
What Quest can be used for:

- **Interactive fiction (text adventures)**. Whereas a book gives you a static world of fiction, Quest lets you create your own world which players can interact with. Instead of having a fixed storyline, players can follow their own path through your story.
- **Computer-based training**. You can simulate any environment using text descriptions, giving you an easy and cost-effective way of developing training materials.
- **Educational software**. Quest can be used to assist pupils and teachers with a variety of subjects:
    - **History**. A teacher can set up a game set in a particular historical period. Pupils can then explore and interact with the objects and characters that appear in the game.
    - **English – creative writing**. Add a new dimension to teaching creative writing by getting your pupils to create an interactive game that other members of the class can play.
    - **Computing**. QDK introduces pupils to a number of programming concepts, such as variables and procedures.

1.2 Quest Components

1.2.1 Quest

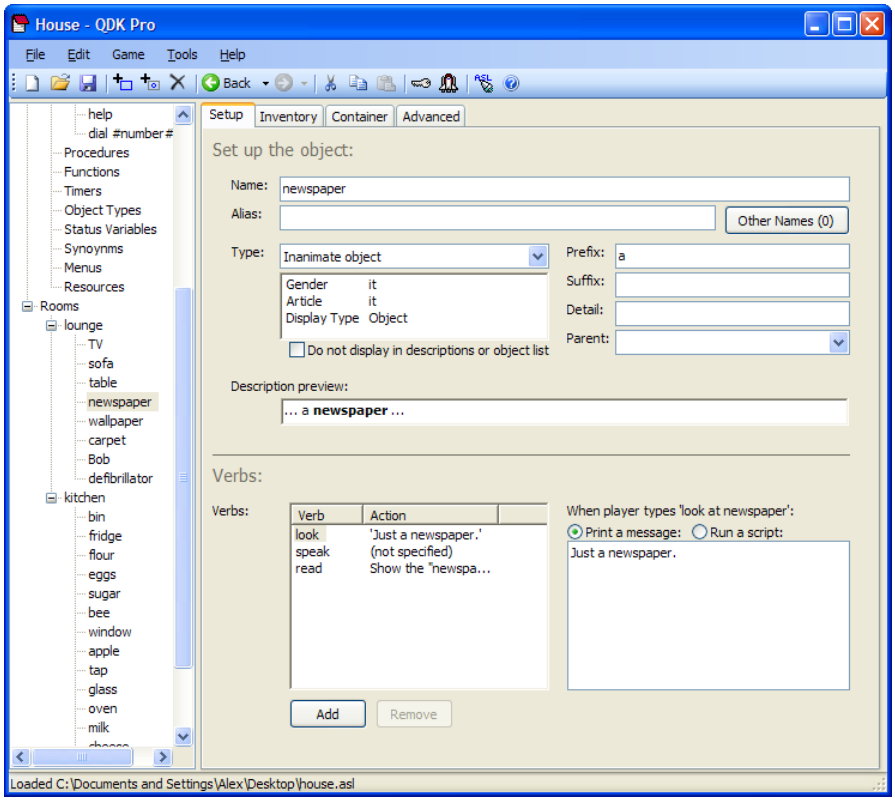
The Quest player lets you play games you have created, or games you have downloaded from a website such as www.textadventures.co.uk.



1.2.2 QDK

QDK stands for Quest Development Kit, and gives you an easy and visual way to create Quest games. You don't need to know how to program – everything about

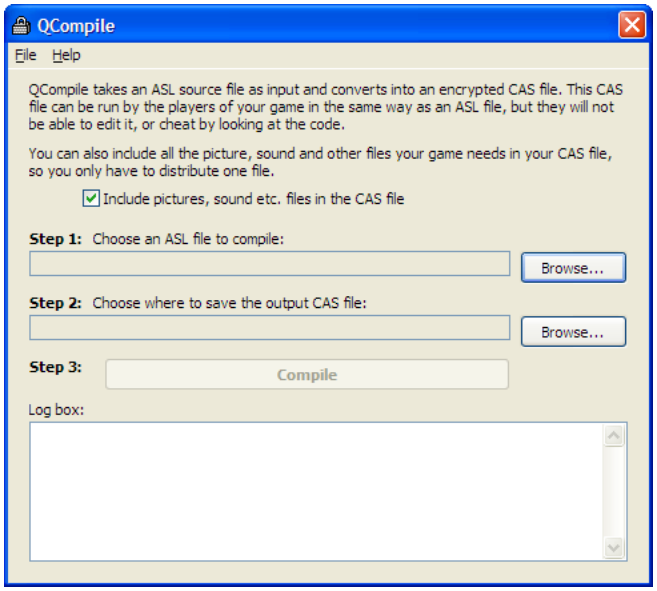
your game is displayed in plain English, and together with this tutorial you've got everything you need to start creating great text adventure games.



The free version of Quest comes with QDK Lite. This is fully featured, except that it is limited to 15 rooms and 50 objects. You can easily complete this tutorial using QDK Lite.

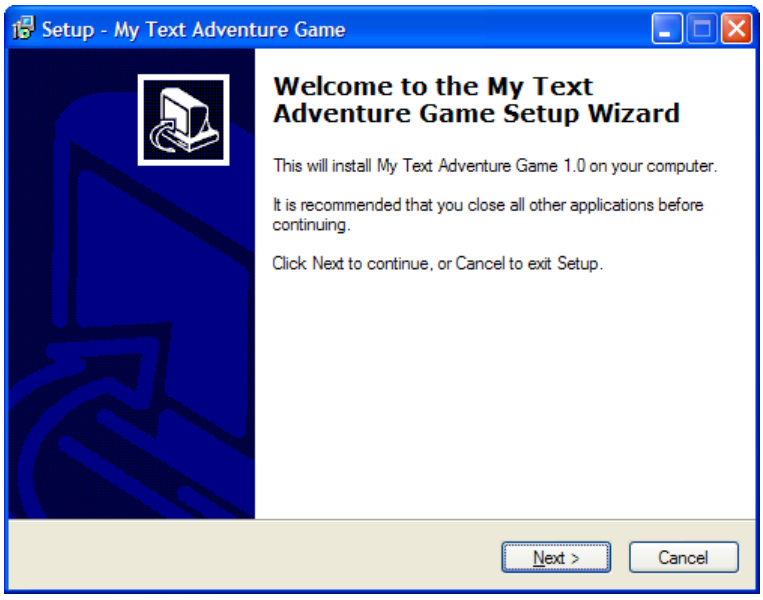
1.2.3 Quest Compiler

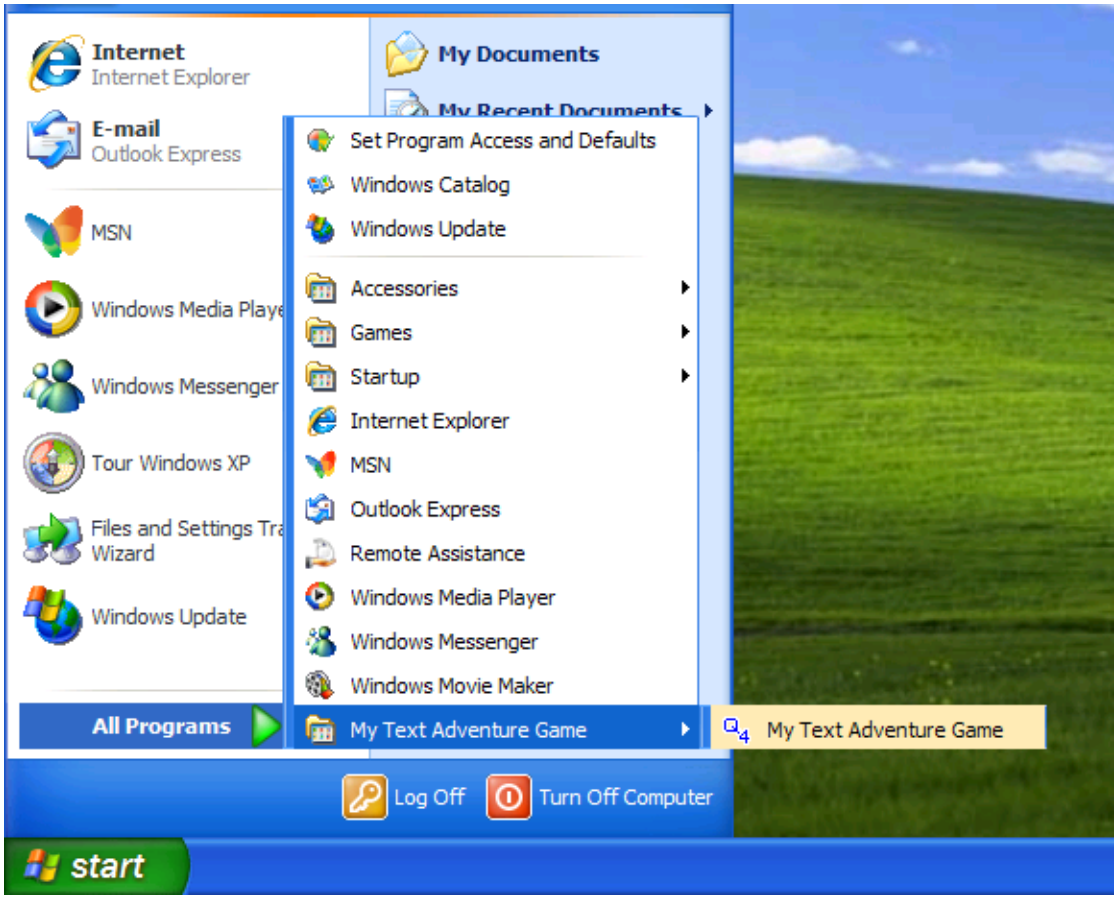
The Quest Compiler lets you take your game file and incorporate it, together with any graphics or sounds your game needs, into just one file. This one file can't be edited by players, so you'll know that nobody can cheat at your game by opening it up in QDK.



1.2.4 Quest Packager

Quest Packager lets you turn your game into a stand-alone setup EXE file. You can then give this setup file to your users, who can install it just like any Windows application. They won't need to download Quest separately, and your game will create its own shortcuts in the Start Menu.





2 The Basics

Quest games are built up from three basic building blocks – rooms, objects and scripts.

2.1 Rooms

Rooms are the locations in your game. They could be rooms in a building, or they could be locations like “the street”, “outside the warehouse”, “the supermarket carpark”, “Jupiter”, or anything else at all.

Each room can have exits to other rooms, which the player can access in two ways:

- **compass directions:** Players can move between rooms by going north, south, north-west etc., either by typing commands or by clicking the directional buttons in Quest;
- **“go to” exits:** Players can move between rooms by typing things like “go to the kitchen”.

2.2 Objects

Objects are the things that exist within your game. This includes inanimate objects like plants, cutlery, newspapers or furniture, and it also includes people or animals within your game.

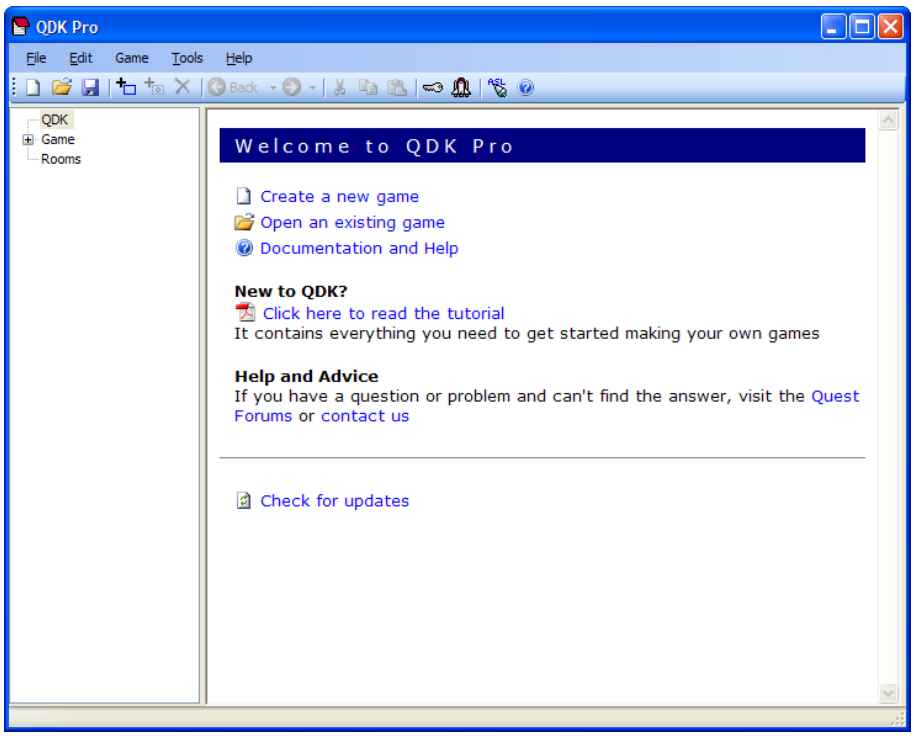
Objects can be looked at, picked up, dropped, spoken to, and more besides – you can set up **verbs** in QDK to do whatever you want them to do.

2.3 Scripts

Scripts are where you set up what actually happens in the game. Does eating the poisoned apple kill the player? If so, it’s a script which makes that happen. Any event that occurs in the game can cause a script to run, whether it’s the player typing in a particular command, or looking at an object, or just entering the room. You can even set up timers so that a script runs at regular intervals. In short, scripts make all the exciting things happen in your game.

3 Setting up the Game

Start QDK. You will see the QDK welcome screen:

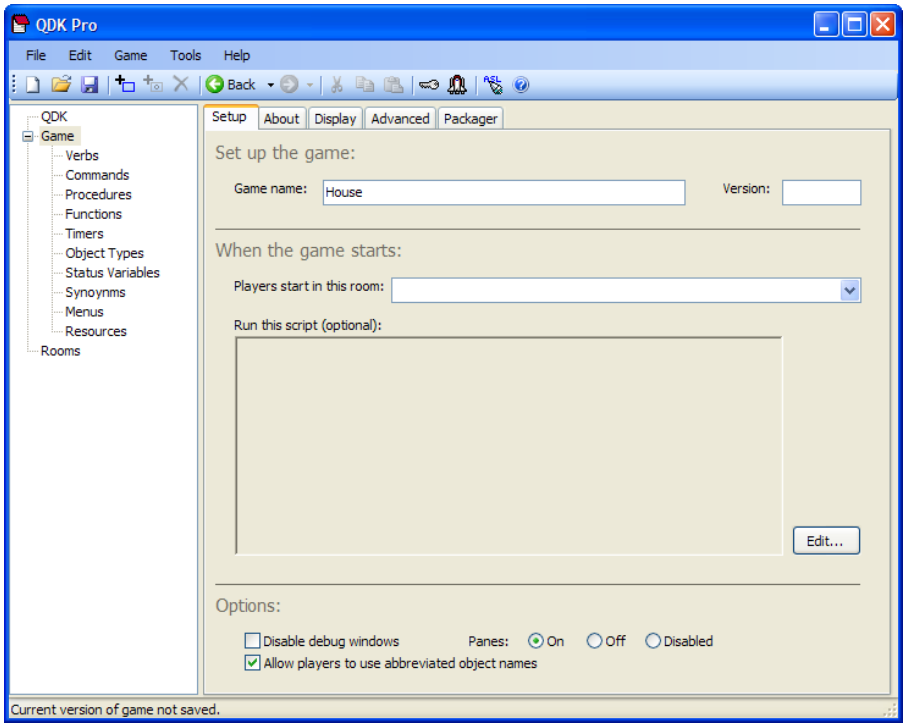


Let's create a new game by clicking the "Create a new game" link.

3.1 Game Properties

After starting a new game, you'll be presented with the Game Properties screen. You can also access this at any time by clicking "Game" on the tree menu at the left of the window. From here you can change the name and version number of your game and more.

We're going to set our game in a house, so you could enter the name "House" – or perhaps think of something slightly more imaginative.



3.2 Adding some Rooms

We need at least one room in the game, so let's add some now.

To do this, select the "Game" menu and click "Add Room", or just click the "Add Room" button on the toolbar.

Add a room called "lounge" and a room called "kitchen".

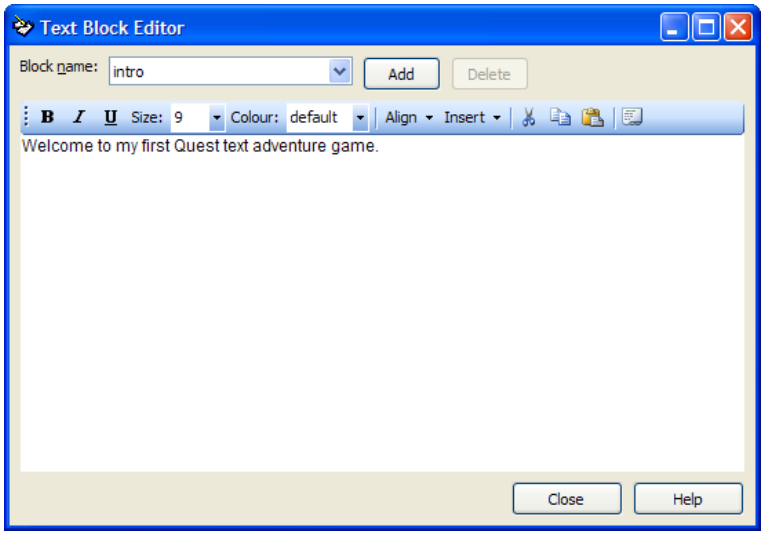
3.3 Adding Introductory Text

Let's add some introductory text to welcome players to the game. To do this, click the Tools menu and then click Text Blocks.

The Text Block Editor window will open, and the "intro" text block will be selected for you. This is where we enter the text that appears when the game begins. There are two other default text blocks, "win" and "lose", which are printed when the player wins or loses the game.

You can also add text blocks of your own for any part of your game where you want a large amount of text to appear. For now though, we're just going to edit the "intro" text block.

Enter some text to welcome players to your game. For the purposes of this tutorial, you could just enter "Welcome to my first Quest text adventure game", though I'm sure you'd prefer to write something a little more original.



When you've finished entering your text, click the Close button to return to the main QDK window.

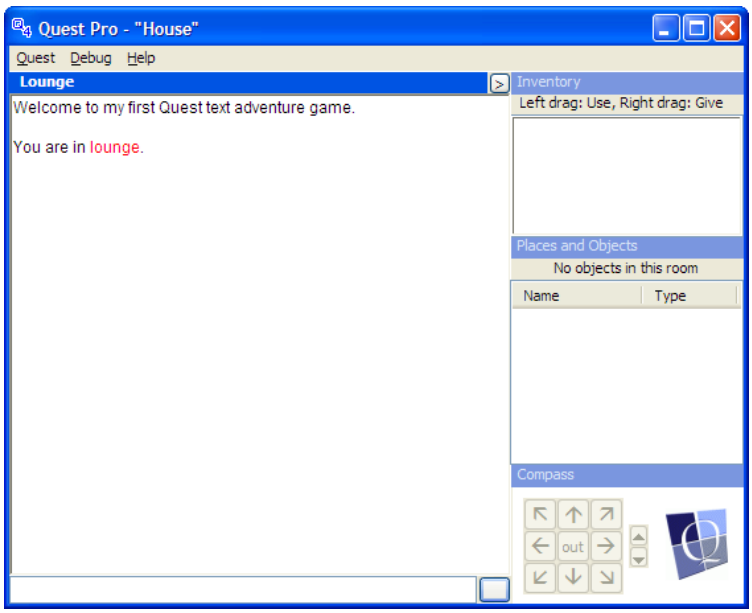
3.4 Launching the Game

Let's now start Quest and see what our game looks like. To do this, you can click the "Run" toolbar button  , or you can click the File menu and select Run. If you're asked whether you want to save the game, click Yes.

Quest will now load your game and you will see something like this:

> Welcome to my first Quest text adventure game.
>
> You are in lounge.

At this point, there's not a lot we can do other than type "look" to show the description again, so close Quest and we'll continue building the game.



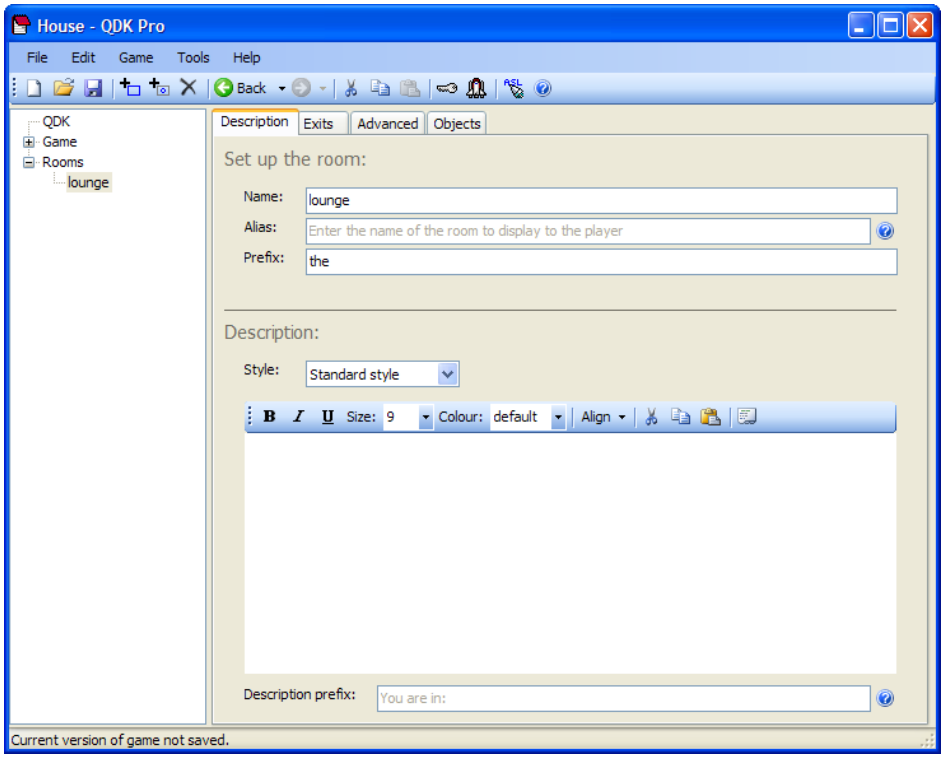
4 Setting up a Room – the Lounge

Now let's set up the lounge. Click "lounge" from the Rooms list on the left side of the QDK window to view the Room properties.

4.1 Setting the Prefix

When we launched Quest, we saw the line "You are in lounge" which isn't quite correct – we want it to say "You are in the lounge".

To do this, we need to set the prefix. This is what appears in front of the room name when Quest tells the player where they are. In this case, we want the word "the" to appear. Enter "the" in the "Prefix" box.



Launch the game again and you will see this line now appears correctly.

> *Advanced tip*: If you want Quest to say something other than the default "You are in…" when the player goes into a room, you can enter some text into the "Description prefix" box at the bottom of the Room properties screen. For example, you might want to use "You are standing outside", "You are waiting at the bus stop" depending on which best fits the type of location you are creating. If you want Quest to automatically add the room name to your description prefix, add a colon ":" at the end.

4.2 Creating a Description

There are three types of description we can use:

- **Standard style** (default). With this type of description, Quest automatically displays the name of the room ("You are in the lounge") and lists all the objects that are in it. If we write a description, this is displayed after this automatic text is printed.
- **Words only**. This type of description displays only the text we write in the box – Quest does not automatically add any other kind of text. This results in a

more traditional style of room description, and if well-written it can give a more immersive experience to the player. However, it will not automatically update if objects or exits in the room change during the course of the game.

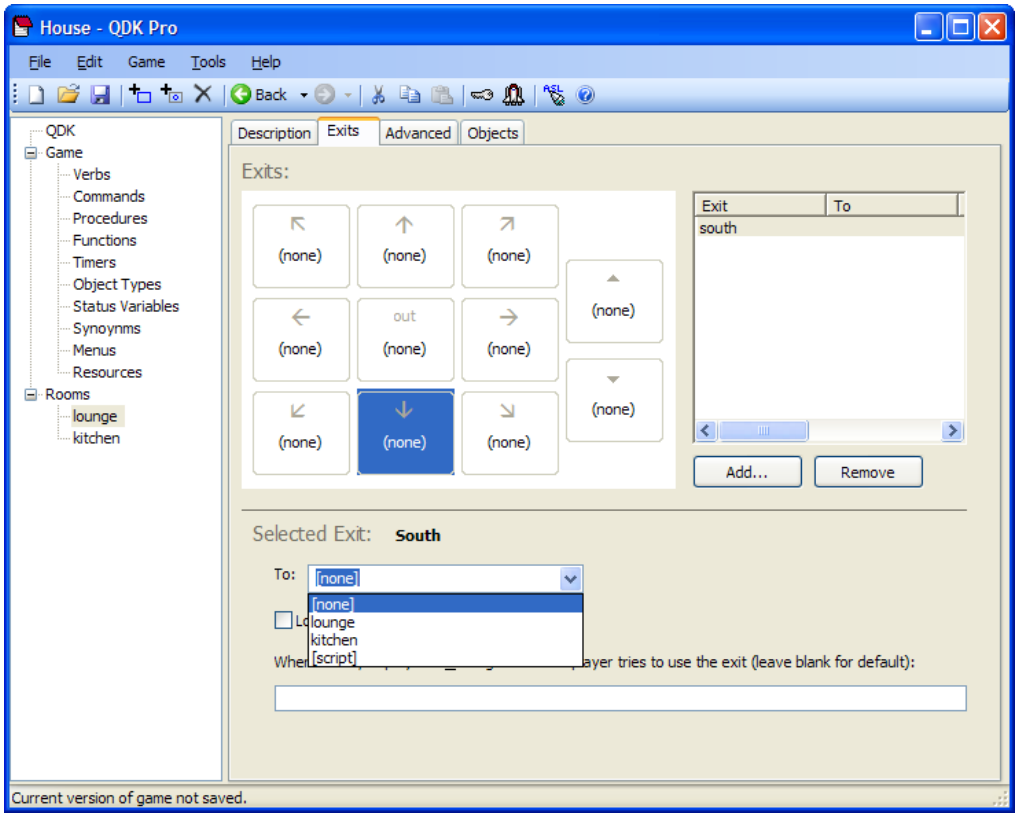
- **Run script**. This is the most flexible type of room description, although it is harder to configure. Instead of printing some text, a script is run. This script can then generate a room description according to the rules you specify.

For now we will use a Standard style description, so you can see how this automatically updates when we add objects and exits to the room. So, ensure "Standard style" is selected from the drop-down Style list, and enter some text into the description box. You could use something like "This is quite a plain lounge with an old beige carpet and peeling wallpaper."

4.3 Adding an Exit

At the moment we have two rooms in our game, the lounge and the kitchen. The player can't get into the kitchen yet though, as we haven't added any exits from the lounge. Let's do this now. Let's have the kitchen to the south of the lounge, so the player can go into the kitchen by typing "south", or by clicking the South direction button.

Go to the Exits tab and select the "South" exit. Look at the options available on the "To:" drop-down.

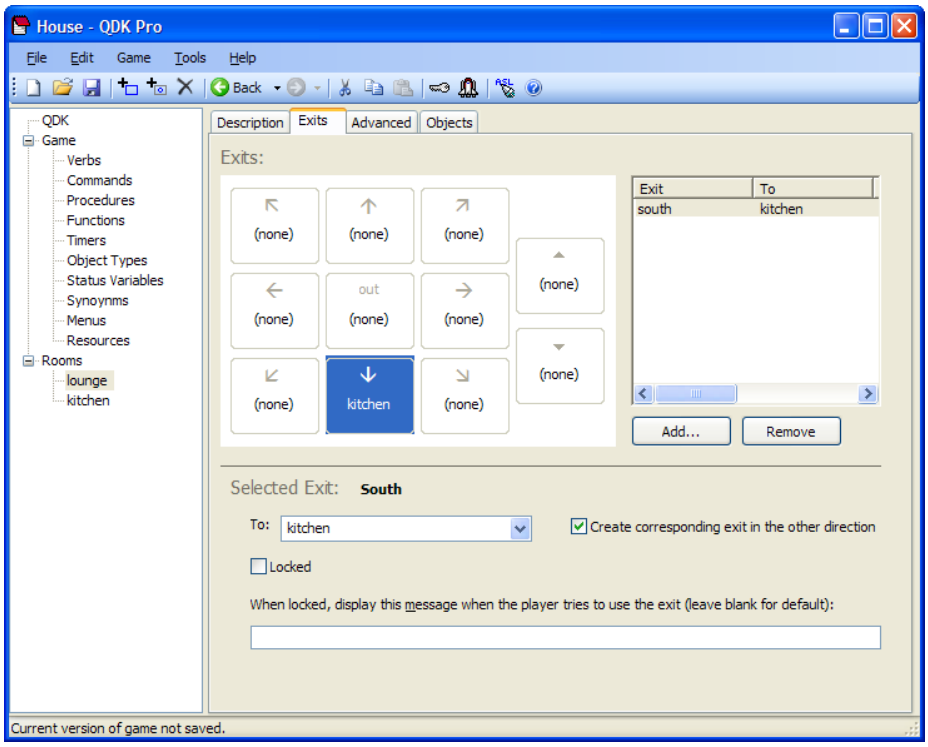


You will see the following options:

- **[none]**. This is the current option. It means that there is no south exit from this room.

- **Lounge**. OK, so we *could* make the south exit come back into the very same room if we wanted to. Let's assume for now that the world our game exists in has a more conventional approach to geometry.
- **Kitchen**. This is the option we want to choose.
- **[script]**. This lets us run a script when the player tries to go south. This is useful if you want to do something different when the player tries to go south, like see if they are carrying a particular item before allowing them to pass.

Select "Kitchen" to create the exit. You will see a checkbox showing that QDK will automatically create another exit for us – the exit that goes north from the kitchen back into the lounge. Keep this selected, as it saves us from having to create this other exit manually.



4.4 Launching the Game

Launch the game again, and you will now see that we have something that is beginning to work like a text adventure game. When you start the game, you should see text like this:

> Welcome to my first Quest text adventure game.
>
> You are in the lounge.
>
> You can go south.
>
> This is quite a plain lounge with an old beige carpet and peeling wallpaper.

If you type "south" or click the south direction button, you will end up in the kitchen. You can then go north back to the lounge.

As an exercise, update the kitchen's prefix and description now.

Our game is not very exciting yet, but at least we've made a start at creating our game world. Now it's time to add some things into these rooms.

5 Creating an Object – the TV

A lounge is barely a lounge without a TV in it, so let's add one now.

5.1 Adding the Object

With the lounge selected, you have four different ways of adding an object to the room. You can:

- Click the Add Object button on the toolbar 
- Click the Game menu and select Add Object
- Right-click the Lounge in the tree menu and select Add Object
- Click the Objects tab and click Add

Use one of these methods to add an object to the lounge. A prompt will appear asking you to enter a name for the object. Enter "TV" and click OK.

5.2 About Object Names

It is important to note the distinction between:

- the names that **players** can use to refer to objects
- the names that your **QDK scripts** use

5.2.1 Name

In order to avoid confusion, each object within QDK must have a unique name. So, if you have multiple televisions in your game, within QDK they must be given different names – like "TV1", "TV2" and so on.

5.2.2 Alias

Of course, this wouldn't sound natural if these were the names that players saw, which is why QDK lets you set an **alias**. This is the name of the object that the player sees. In the example of multiple televisions, each of your TV objects could have an alias of "TV".

If you don't set an alias, players will see the same object name that you use in QDK – so you only usually need to worry about this if you want different objects to have the same name.

For now then, we will leave the Alias box for our TV object blank.

5.2.3 Other Names

The "Other Names" box lets you specify *additional* ways that players can refer to this object. It is important to note that different players will have different ways of interacting with your game – some players prefer to use the lists that Quest displays on the right-hand side of the window to interact with objects and so on, but many players prefer to use good old-fashioned commands. You want to make it easy for Quest to understand what players type in, so you can add additional, alternative object names to ensure that happens. For example, for our TV object, some players might type in "look at television", and would reasonably expect that to work.

Click "Other Names" and add "television" to the list of names for this object. This will ensure that players can type in either "look at TV" or "look at television" to look at this object.

As an exercise, add any other alternative names you think that players might want to use.

5.2.4 Summary

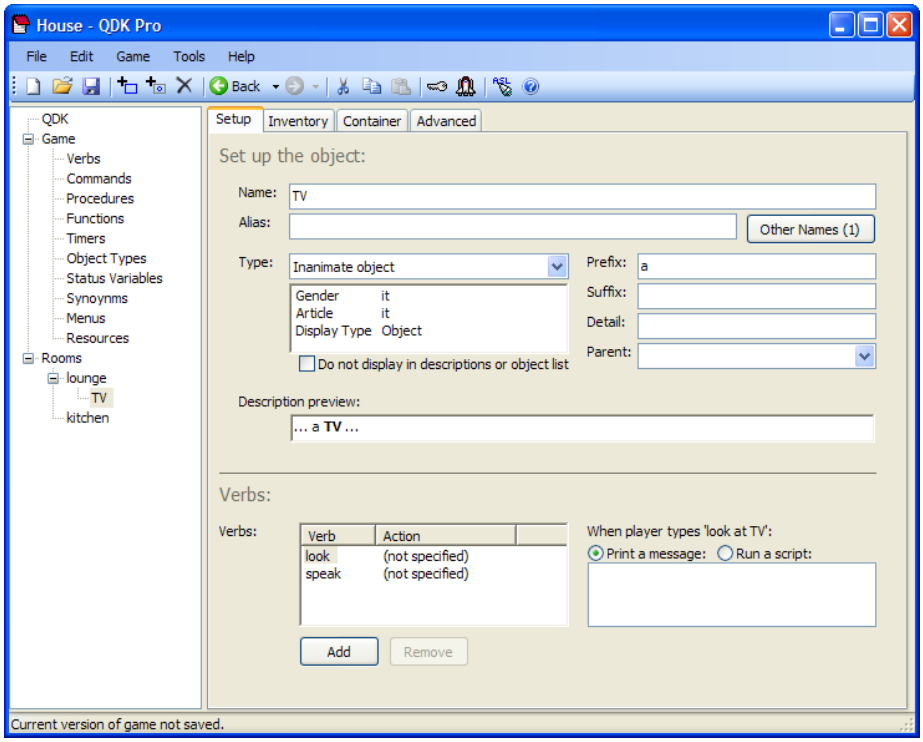
Name	Alias	Other Names
• Must be unique • The name that your QDK scripts use internally • Only one name allowed for each object	• Doesn't have to be unique • The name that players see on the screen • Only one alias allowed for each object	• Doesn't have to be unique • Extra names that players can type in • As many additional names as you like

5.3 Setting the Prefix

If you run the game now, you should see something like this:

> Welcome to my first Quest text adventure game.
>
> You are in the lounge.
>
> There is TV here.
>
> You can go south.
>
> This is quite a plain lounge with an old beige carpet and peeling wallpaper.

As we did when setting up the room, we need to add a **prefix** to this object to make the description sound natural. We want Quest to say "There is a TV here", so type "a" in the Prefix box. Launch Quest again and verify that it displays the description correctly. You should also see that QDK's Description Preview box updates accordingly.



5.4 Suffix and Detail

The **suffix** appears *after* the object name in the room description. You can use this for a brief description of the object that will be displayed when the player enters the room. For example, if you entered a suffix of “showing an old western” for the TV object, the room description would show “There is a TV showing an old western here”.

The **detail** is used if you have several objects called the same thing. If you had several TVs in the lounge, with their aliases all set to “TV”, the detail is what’s shown in the menu that Quest displays when it asks the user to pick which TV they want to refer to.

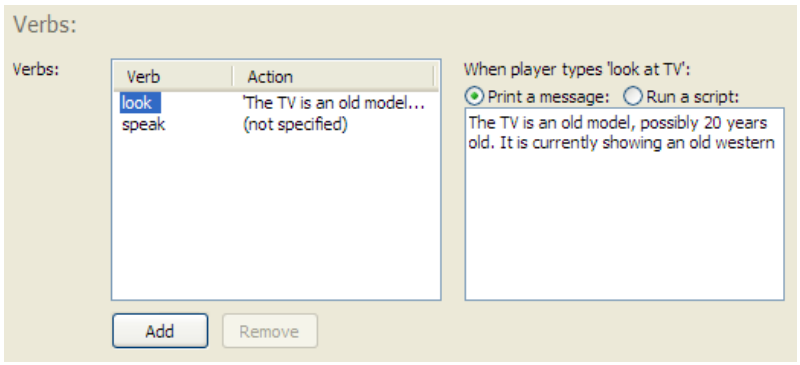
5.5 Setting the Description

If you run the game and type “look at tv” (or, remember, you can use “look at television”) you’ll see that Quest doesn’t have much to say on the subject:

> Nothing out of the ordinary.

Let’s add a description for this object. To do this, go to the “Verbs” list in the bottom half of the Object’s Setup screen and select “look”. You’ll see we have two options – we can either just print a message, or we can run a script when the player looks at the object.

For now, let’s just print a message describing the television. You could write something like “The TV is an old model, possibly 20 years old. It is currently showing an old western.”



Launch the game again and verify that it now shows you the description when you look at the TV.

5.6 Adding a Verb – Watch the TV

It is a good idea to think about what kinds of things players might try to do to any objects in your game. In our example of the TV, it seems likely that a player might try to “watch tv”, so it would be good if our game came up with a good response, rather than just saying it didn’t understand.

To do this, let’s add the **verb** “watch” to our TV object. As you should remember from school, verbs are “doing words”, and that’s what they are in Quest – verbs let you specify what things can be “done” to your object.

Click the “Add” button underneath the Verbs box, and type “watch”. Just as with the “look” description earlier, we can either print a message or run a script when the player watches the TV. Enter a message. For example, “You watch for a few minutes. As your will to live slowly ebbs away, you remember that you’ve always hated watching westerns.”

5.7 Exercises

As an exercise, add the following objects to the lounge:

- A sofa. Give it a prefix of “a” and enter a sensible description. Add a verb “sit on” so that the player can type “sit on sofa”. This should print a message like “There’s no time for lounging about now.”
- A table. Enter a sensible prefix and description.
- A newspaper. Enter a sensible prefix and description. Add a verb “read” and enter an appropriate message.

Launch the game and verify that the objects you’ve just created have been set up and are working correctly – check that you can watch the TV, try to sit on the sofa, and read the newspaper. Also check that the room description is displaying correctly – it should say something like “There is a TV, a sofa, a table and a newspaper here” if you have set up all the prefixes correctly.

5.8 Points to Note

Notice that by adding verbs to an object, we automatically set up that verb in the game. So, when you set up a “read” verb for the newspaper, you made Quest automatically understand the command “read”. At the moment, the player can only read the newspaper, but notice that if you type something silly such as “read table”, Quest says “You can’t read that”. Before you set up the verb “read”, Quest would have responded “I don’t understand your command”, but now it understands the word “read” it gives a much more sensible response (and we’ll see later on how to customise this).

We’re now on our way to making our first text adventure game. You may have noticed that, so far, we’ve only been walking around the game world and looking at things – we’ve not yet managed to interact with it and change it. We’ll start to do that in the next chapter, where we look at taking and dropping objects.

6 Ways with Objects

6.1 Different Types of Object

So far, all of our objects have been inanimate, and they've all appeared in the description of our room and in Quest's "Places and Objects" list. QDK lets you specify an object type from the following options:

- Inanimate object
- Inanimate object (plural)
- Male character
- Female character
- Scenery
- Custom

Whatever type you select, the object will pretty much behave the same way in Quest. You still get all the same options – specifying a description, adding verbs and so on – but Quest's default responses will make a lot more sense if this option is set correctly. This is because setting the type will update the following settings:

- **Gender**. Usually "it", "he", "she" or "they". Quest uses this, for example, as the default response when the player tries to speak to the object, so it will display "*It* says nothing", "*He* says nothing" or "*She* says nothing" as appropriate.
- **Article**. Usually "it", "him", "her" or "them". Quest uses this, for example, as the default response when the player tries to give an object to another object, so it will display "He doesn't want *it*" or "He doesn't want *them*" as appropriate.
- **Display Type**. Usually "object" or "character", but this can be whatever you want. This is used in the "Places and Objects" list when the game is played in Quest.
- **Do not display in descriptions or object list**. This is set when you select "Scenery". This option means that the object won't display in Quest's "Places and Objects" list or in the automatically generated room description (if you're using the "Standard style" description).

6.2 Creating a Scenery Object

Let's create a scenery object. Why might we want to do this? Well, when we made our room description we wrote "This is quite a plain lounge with an old beige carpet and peeling wallpaper". What if the player types "look at wallpaper"? Quest will reply "I can't see that here", which breaks the flow of our game somewhat.

Although the wallpaper isn't an important object, we should still have a response for "look at wallpaper", so we should set it up. If we make it a scenery object though, it's "in the background" as far as the game goes – since it doesn't appear in the "Places and Objects" list, or in the list of objects in the description of the room, we won't be cluttering things unnecessarily.

So, create a new object called "wallpaper". Set the type to "Scenery". We don't need to set a prefix or suffix since this won't appear in our room description. Enter a description like "The horrible beige wallpaper hangs loosely on the walls."

Launch the game and verify that although the wallpaper doesn't explicitly appear in the description, you can still get a sensible response by typing "look at wallpaper".

Remember that we also mentioned the carpet in the room description as well. As an exercise, add this as another scenery object, and give it a sensible description.

6.3 Creating a Character

Let's create our first character. He'll be about as basic a character as you can get, and he won't be the most talkative. This is because he's dead. Well, you've got to start somewhere.

Create a new object called "Bob" and change its type to "Male character". Give him a "look" description of "Bob is lying on the floor, a lot more still than usual." We will come back to Bob in chapter 12, where will make him a little more animated.

7 Adding a Script – Switch off the TV

We'll now start to play with the real power behind Quest – scripts. Scripts let us do things within the game, change the game world, show pictures and more. In this example, we'll make a simple script that allows the player to switch the TV off.

We covered verbs earlier, and we'll set up a verb that allows the player to type "switch off tv". Up until now, our verbs have just printed text – now, however, we want the game world to change when the TV is switched off. We'll need to change what happens when the player types "look at tv" or "watch tv", because our current descriptions refer to the TV being switched on. Fortunately, it's easy to change these using script commands.

Here's what we want to happen:

- Before switching the TV off, the "look" and "watch" descriptions should remain the same as before, i.e.:
    - Look: "The TV is an old model, possibly 20 years old. It is currently showing an old western."
    - Watch: "You watch for a few minutes. As your will to live slowly ebbs away, you remember that you've always hated watching westerns."
- After switching the TV off, the "look" and "watch" descriptions should change, to something like this:
    - Look: "The TV is an old model, possibly 20 years old. It is currently switched off."
    - Watch: "You watch for a few minutes, thinking that the latest episode of 'Big Brother' is even more boring than usual. You then realise that the TV is in fact switched off."

There are a number of ways you could accomplish this:

- You could run a script in the "switch off tv" verb to change the "look" and "watch" **properties** of the TV object.
- You could run a script in the "switch off tv" verb to set a **flag** which tells Quest that the TV is switched off. Then you could run a script in the "look at tv" verb to print the relevant description depending on whether the flag was set, and likewise run a script in the "watch tv" verb.

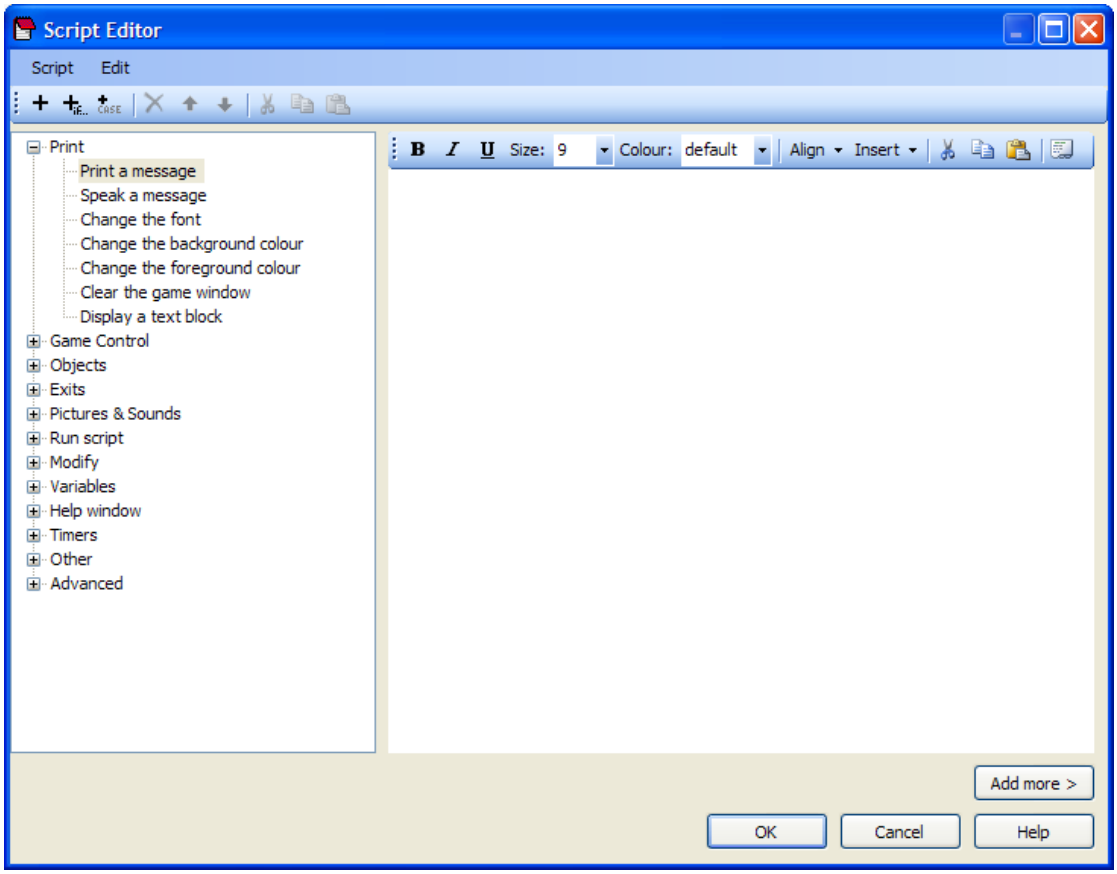
For now we will use the first approach, setting properties. We will look at flags later on.

7.1 What are Properties?

Properties allow you to store custom information about an object – you could use them to store information about an object's weight, value and so on, which you can then refer to within your scripts. They also provide a way for you to change things like an object's initial description, and it's this feature that we'll be using here.

7.2 About the Script Editor

Select the TV object and add the verb "switch off". Select the "Run a script" option and then click the "Edit" button that appears. You will now be presented with the Script Editor.



On the left hand side of the Script Editor is a tree menu which shows you all of the available script commands. They are grouped into categories of related commands. When you select a command, the box on the right will change to let you enter any information that the command requires. It may also provide you some helpful information on what the command is for, and how to use it.

At the top of the Script Editor is a toolbar which allows you to add additional commands to the script, delete them, change the order, and cut/copy/paste. It also lets you add a couple of other types of script – a **conditional** and a **select case**, which we'll cover later.

7.3 Adding the Script Commands

We're going to add three script commands which will run when the player types "switch off tv":

- The first script command will change the TV's "look" description
- The second script command will change the TV's "watch" description
- The third script command will tell the player that they have switched off the TV.

We'll add these commands one at a time.

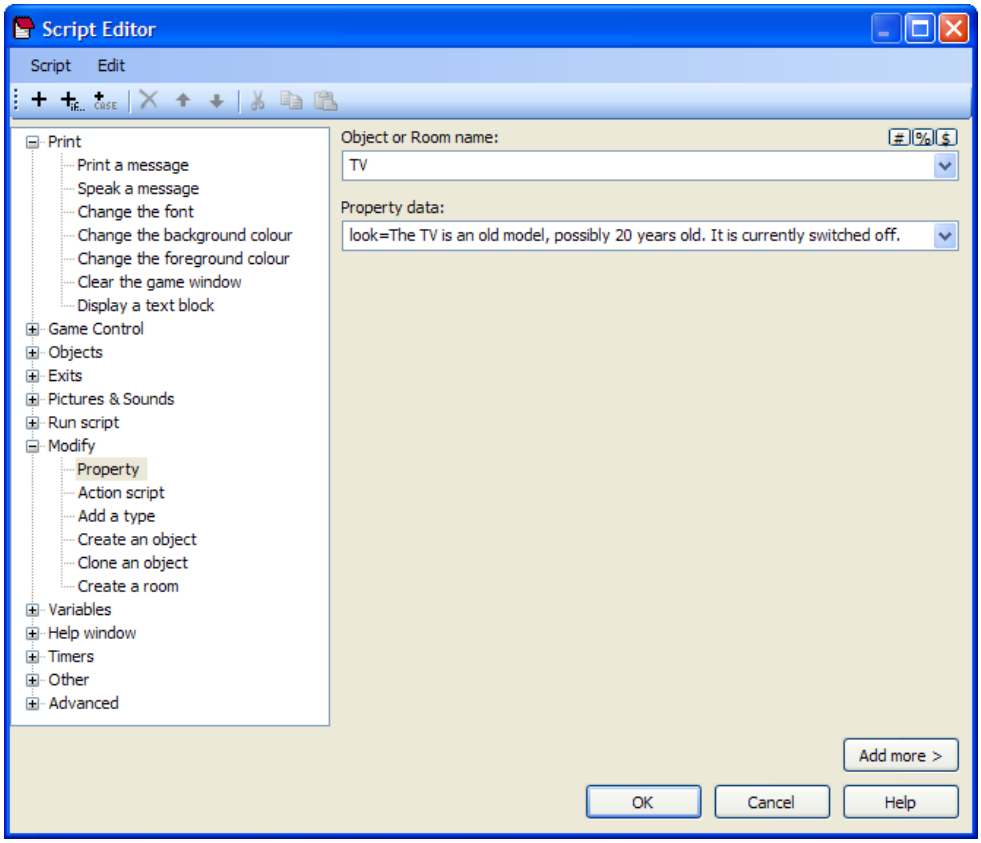
7.3.1 Command 1 – "look" Description

To add the first command, double-click the "Modify" category to show the commands within it. Select the first command, "Property".

On the right-hand side, you will see options for selecting the object you want to modify, and a box to enter the property data you want to apply.

Click the “Object or room name” drop-down list and select “TV”.

In the “Property data” box, enter “look=The TV is an old model, possibly 20 years old. It is currently switched off.”



Click OK.

You will see that the script is entered underneath “Run a script” and it looks like this:

> Modify "TV"'s property: "look=The TV is an old model, possibly 20 years old. It is currently switched off."

Launch Quest now and type “switch off tv”. You won’t see a response to this command printed, because we haven’t added that yet. But, type “look at tv” afterwards and verify that the description has indeed been updated.

The Quest output should look something like this:

> > look at tv
>
> The TV is an old model, possibly 20 years old. It is currently showing an old western.
>
> > switch off tv
>
> > look at tv
>
> The TV is an old model, possibly 20 years old. It is currently switched off.

7.3.2 Command 2 – “watch” Description

To add the second command, click Edit to return to the Script Editor. To add a second command, either click the “Add Command” button on the toolbar + or click the “Add more >” button on the bottom-right of the window.

The command we'll use is the same as in command 1, but this time we'll enter some different property data. Open the "Modify" category again and select "Property". Choose the "TV" object, and in the "Property data" box enter "watch=You watch for a few minutes, thinking that the latest episode of 'Big Brother' is even more boring than usual. You then realise that the TV is in fact switched off."

Click OK. The script should now look like this:

```
Modify "TV"'s property: "look=The TV is an old model, possibly 20 years old. It is currently switched off."

Modify "TV"'s property: "watch=You watch for a few minutes, thinking that the latest episode of 'Big Brother' is even more boring than usual. You then realise that the TV is in fact switched off."
```

Launch the game and verify that Quest gives the correct response to "watch tv" after you switch it off.

7.3.3 Command 3 – Print a Message

We'll add one final command so that the player sees some output when they type the "switch off tv" command. Go back into the Script Editor and add a new command. From the "Print" category, select "Print a message". Enter the message "You switch the TV off." and click OK.

The finished script should now look like this:

```
Modify "TV"'s property: "look=The TV is an old model, possibly 20 years old. It is currently switched off."

Modify "TV"'s property: "watch=You watch for a few minutes, thinking that the latest episode of 'Big Brother' is even more boring than usual. You then realise that the TV is in fact switched off."

Print "You switch the TV off."
```

7.4 Adding Verb Alternatives

We've now successfully added "switch off tv" as a valid verb, and told Quest what to do when a player types that in. But what if the player types "turn off tv" instead?

With the Additional Verbs library turned on (which it will be, unless you've turned it off), Quest will recognise "turn off tv", but it will say that the player can't turn the TV off – because we haven't told it otherwise. And with the Additional Verbs library turned off, Quest won't recognise the command at all.

This kind of "guess the verb" is something to be avoided – as a game author, you need to think about all the different ways a player might want to express something.

It would be a pain if we had to add all those script commands to a new "turn off" verb though, and fortunately we don't need to do this – we can simply update our existing "switch off" verb so it also works with "turn off".

Select "Verbs" on the tree menu on the left side of the QDK window. You will see that QDK lists here all the verbs we have set up in our game. Select "switch off" and click Edit.

The "verb" box currently says "switch off". Update this so it reads "switch off; turn off".

Launch Quest and verify that "turn off tv" now has the required effect.

How does this work? If you specify multiple commands in a verb list, Quest will treat them all as the same thing when the player types them. The first verb in the list is the primary verb, in this case "switch off", and this is the name that is used internally within QDK when setting up your objects. "Turn off" is treated as a synonym of "switch off", so it runs the same script when typed by the player.

8 Interacting with Objects

In this chapter, we'll cover picking up and dropping objects.

At the moment, when the player types "pick up newspaper" or "take tv", they will be told "You can't take it". You need to set up objects so they can be taken by the player, but fortunately this is easy to do.

8.1 Taking the Newspaper

Let's make the newspaper take-able. Select the newspaper object from the tree menu, and click the Inventory tab. In the "Take" section, select "Yes" to allow this object to be taken.

Launch the game now and verify that you can take the newspaper. Quest's default response is "You pick it up". If you want to change this, you can just enter your own message next to where you selected "Yes" in the "Take" section. Enter something like "You put the newspaper in your back pocket."

8.2 Trying to Take the Sofa

You can also run your own script when the player tries to take something. You could use this, for example, so that the player could only take something if they'd already completed some other action in the game (for this you would need to use a conditional script – see section 9).

Note that when you use the "Run a script" option, the object will *not* automatically be moved into the inventory. To move the object into the inventory, you need to do this within your script by adding the "Give an object to the player" command from the "Objects" category.

Here, we will use a script just to print a message when the player tries to pick up the sofa. All we need to do is include the "Print a message" command, as using a script ensures the object will not automatically be moved into the inventory.

Select the sofa object, and then go to the Inventory tab. Select the "Run a script" option from the "Take" section and click Edit to bring up the Script Editor. The "Print a message" command is selected for you, so enter the message "You're not quite strong enough to pick up the sofa by yourself."

8.3 Dropping Objects

By default, once an object has been picked up, it can be dropped everywhere. You can turn off drop entirely by selecting "Nowhere" from the drop options. Let's try this now. Select the newspaper object and go to the Inventory tab, then select "Nowhere" from the Drop options. Enter a message such as "Best not put this down – you might need this quality journalism later on."

Launch the game and verify that you can no longer drop the newspaper after taking it.

As with taking an object, you can also specify a script when the player tries to drop. This gives you total flexibility – when you use a script, the object will *not* automatically be dropped, so you can use conditional scripts together with the "Take an object from the player" script command (again, in the "Objects" category) to allow the player to drop an object only in certain locations, for example.

9 Into the Kitchen – Flags and Conditional Scripts

We'll now start creating things in the kitchen, where we'll look at some of Quest's more advanced features.

If you haven't done so already, add a sensible prefix to the kitchen. We'll use a standard style description for now – enter a description like "The kitchen is cold and the stench of the overflowing bin makes you feel somewhat faint."

As an exercise, add a scenery object called "bin" and give it a sensible description.

9.1 About Flags

In section 7.3, we looked at changing the description of something during the game by modifying its "look" property (and our custom "watch" property).

We'll now look at another way of doing this, using **flags**. Flags let us specify if something has happened in our game. We can then use script commands to check whether these flags are **on** or **off**, so we can then instruct Quest to, for example, print the correct description. In this example, we'll be using a flag to tell Quest whether the oven is on or off.

The advantage of this approach compared to the one we used before is that we can easily toggle the state – in this example, we'll be able to easily switch the oven on and off from within our script, without having to change all of the properties each time. It also makes it easier for other pieces of script to tell whether the oven is on or off.

9.2 Creating the Oven

First, create an object in the kitchen called "oven", of the default "Inanimate object" type and with a prefix of "an".

The first thing to do is choose a name for our flag. In this example, we'll call it "oven on". It is important to remember that all flags are **off** to begin with.

The description of our oven will depend on whether the "oven on" flag is on or off:

- If the flag is **off**, the description will say "The old oven is caked in dirt."
- If the flag is **on**, the description will say "The old oven is on, and you can just make out the light inside through the dirty glass of its door."

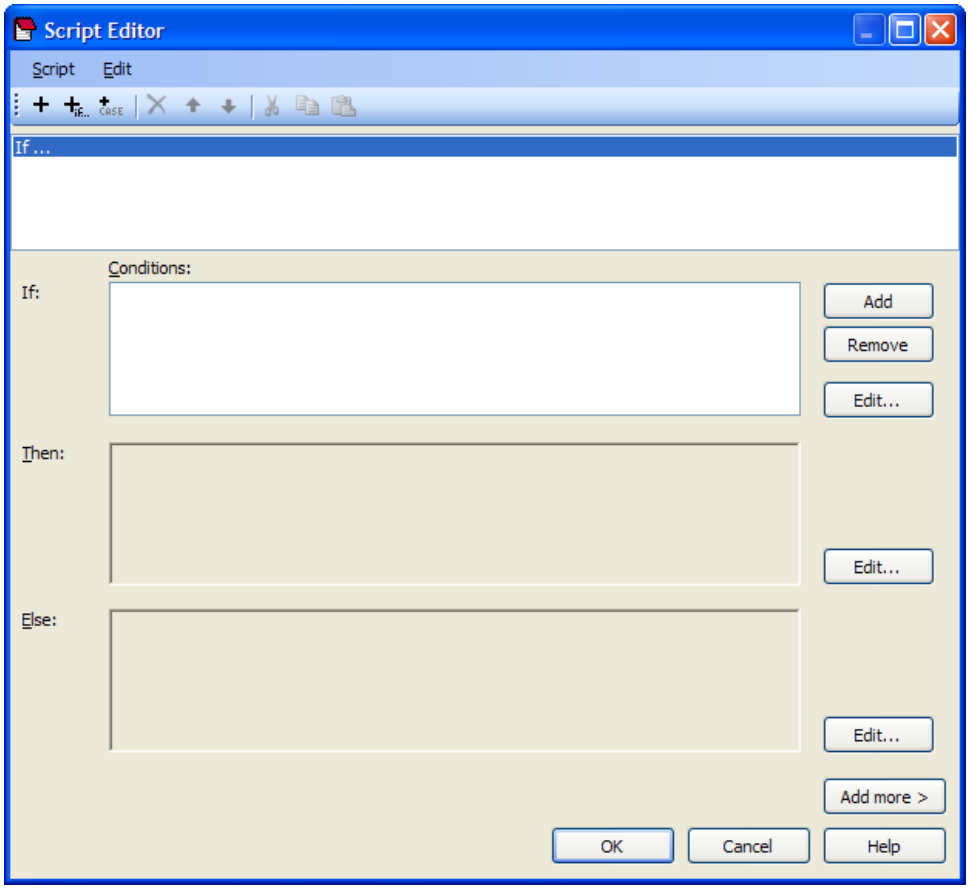
We want to put some script into the oven's look description which will first check to see if the "oven on" flag is on, and then print the correct script. Quest lets us do this using **conditional scripts**.

9.3 Creating a Conditional Script

Select the "oven" object and then select the "look" verb. Select "Run a script" and then click the Edit button to show the Script Editor.

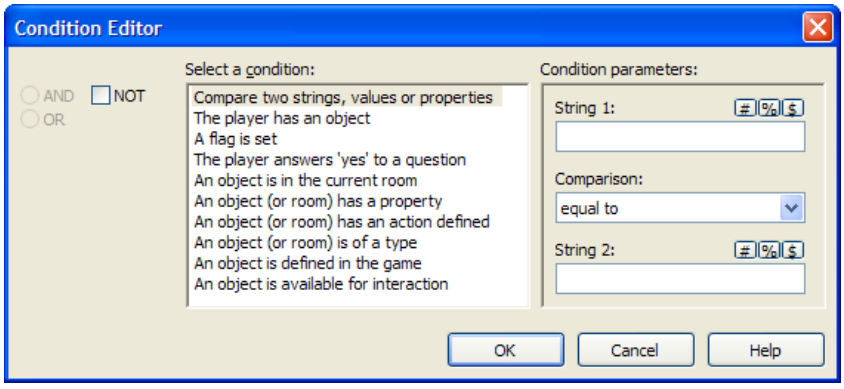
The Script Editor will show, ready for you to enter an ordinary script command. We want to enter a conditional script though, so click the "Add Conditional" button on the toolbar  .

The Script Editor window will now change to give you a list of Conditions, plus two further script boxes which allow you to specify your "Then" and optional "Else" scripts.



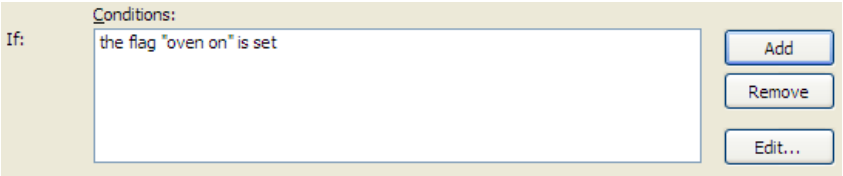
How does this work? The list of conditions specifies what must be true in order for the “Then” script to run. In our example, we’ll use a condition which asks “is the ‘oven on’ flag on?”. If it is, i.e. the flag is on, the “then” script will run. If it is not, i.e. the flag is off, the “else” script will run (if we enter one – otherwise nothing will happen).

Click the “Add” button next to the Conditions list to add a condition. We will only be specifying one condition here, and it will be the check to see if our “oven on” flag is set. The Condition Editor window will appear.



From the list of possible conditions, select “A flag is set”. On the right-hand side of the screen, type in “oven on”. Click OK.

The condition will be added to the conditions list:



Now we want to enter the relevant script to run if the condition is true – in this case, the description to show when the oven is on. Next to the “Then” script box, click the Edit button.

A new Script Editor window will appear, with the “Print a message” command selected for you. In the Message box, type in “The old oven is on, and you can just make out the light inside through the dirty glass of its door.” Then click OK.

We want to specify an “Else” script (otherwise we will get no description when the oven is off), so click the Edit button for the “Else” script, enter the message “The old oven is caked in dirt.” and click OK.

You have now finished entering the description script, so click OK to close the Script Editor.

If you have set everything up correctly, the script should appear as follows for the oven’s “look” description:

```
If the flag "oven on" is set  Then Print "The old oven is on, and you can just make out the light inside through the dirty glass of its door." Else Print "The old oven is caked in dirt."
```

9.4 Setting the Flag

We have now successfully created a script that will show the correct description depending on if the “oven on” flag is set. We now need to add some script to set this flag when the player types “switch on oven”. We saw how to do this earlier – all we need to do is set up a “switch on” verb for the oven.

With the “oven” object selected, add a verb “switch on” and select the “Run a script” option. Click Edit to open the Script Editor. From the “Variables” category, select the “Set a flag on” command. Enter the flag name “oven on”.

Click “Add more”. Use the correct command to print a message telling the player that they have switched on the oven.

Your script should look something like this:

```
Set the flag "oven on" to on

Print "You switch on the oven."
```

Of course, we also need to set up a verb to switch off the oven. As an exercise, add this verb and make the script turn *off* the flag “oven on”, and print a sensible message.

9.5 Launching the Game

Launch the game and go south to the kitchen. Look at the oven and verify that it displays the correct description (the oven should be off when the game starts). Switch on the oven, and then look at it again. Verify the correct description is displayed.

Your game output should look something like this:

```
> look at oven

The old oven is caked in dirt.

> switch on oven
```

You switch on the oven.

> look at it

The old oven is on, and you can just make out the light inside through the dirty glass of its door.

10 Weighing Things, Part 1 – Using Custom Properties

We've already taken a look at properties in section 7, where we used them to change the "look" and "watch" descriptions of an object during the game. In this section, we'll look at another use of properties – storing custom types of information about an object. In the example we'll look at, we'll store the weights of various items as properties, and in the next chapter we'll set up a "weigh" command which will tell us the weight of an object.

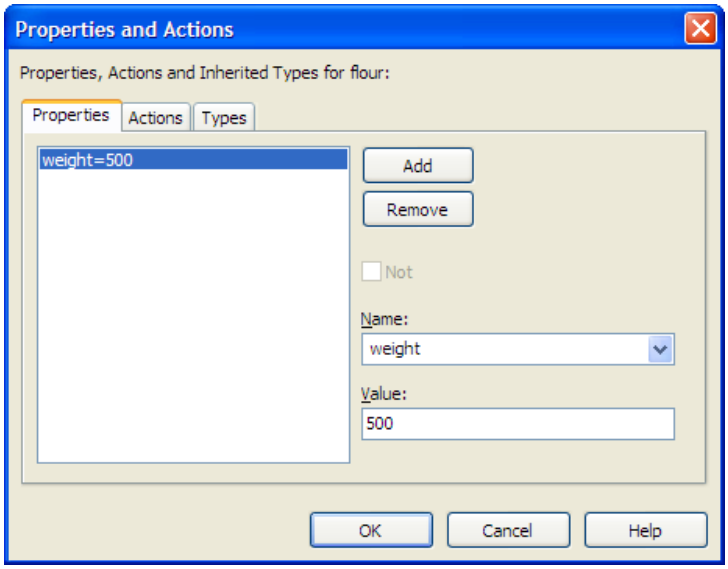
10.1 Adding Custom Properties

First, let us create a few objects we can weigh. Create three objects – flour, eggs and sugar. Give them sensible prefixes, and make sure the object type is set correctly (i.e. either "inanimate object" or "inanimate object (plural)").

Now let's give them some weight. We'll use units of grams so we'll say the flour has a weight of 500, the eggs have a weight of 250, and the sugar has a weight of 1000.

First we'll set the flour's "weight" property to 500. To do this, select the flour object and go to the Advanced tab. Click the "Edit Properties and Actions" button. The Properties and Actions window will now appear, and here we can enter as many of our own custom properties, actions and inherited types as we want. We'll look at actions and inherited types later.

For now, on the Properties tab, click the Add button. Enter the name "weight" and enter the value "500". The list will update to show "weight=500".



Click OK.

As an exercise, follow the same process to set the weights of the eggs and the sugar.

10.2 Reading Custom Properties

We've now successfully set up our flour, eggs and sugar objects with custom information about their weight. We'll now look at how to read this information from within the game.

You can read a property from any script command by typing:

```
#object:property#
```

For example, to read the weight of the eggs, you would type:

```
#eggs:weight#
```

Let's update the "look" description of the eggs as an example. Select the eggs object, and then select the "look" verb within the object properties. Select the "Run a script" option and print the message "A box of eggs, weighing #eggs:weight# grams."

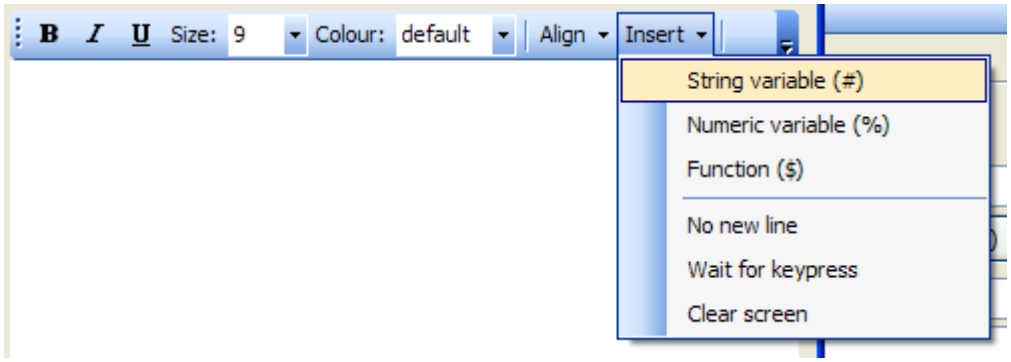
Why didn't we just use the "Print a message" option? Because "#eggs:weight#" needs to be calculated by Quest when it is displayed, so we can't use a static message – remember, it only works from within a script command.

Launch the game and verify that the correct response is displayed – it should read "A box of eggs, weighing 250 grams."

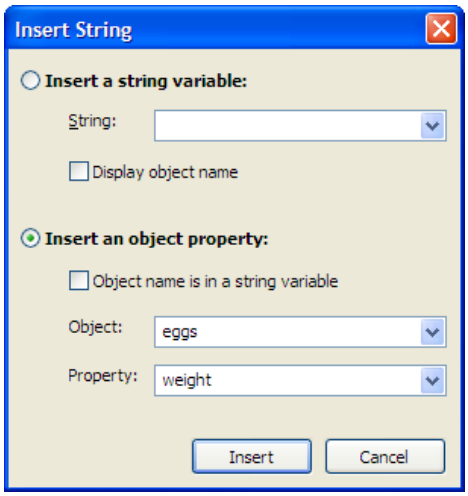
Of course, we could have just manually entered this into the description of the eggs anyway – we didn't really need to use a property. The real power of this, though, is that you can easily change properties while the game is running. We saw this before, in chapter 7, when we changed an object's "look" and "watch" properties. We can do exactly the same thing with these custom properties, so we could change the "weight" property of the eggs during the game – for example, if the player used some of them to bake a cake (hopefully you'll have some better ideas for fun things that players can do in your game). When the property is updated, our "look" description would automatically reflect the current weight of the eggs.

10.3 Using the Insert String window for Properties

Instead of typing in "#eggs:weight#", you can easily insert this by clicking the "Insert" button and selecting "String variable (#)":



You will see the following window, which lets you select the object and property from a drop-down list:



For script commands other than “Print a message”, you can access this window by clicking the “#” button you see above command parameters.

11 Weighing Things, Part 2 – Setting up a Command

In this chapter, we will add a **command** that lets the player type “weigh sugar”, “weigh eggs” etc.

Note that we will *not* be using a **verb** here, as we have done before. Why not? Verbs are good when you want to have a *separate* response for each object – for example, when we set up our first “watch” verb, we wanted a particular response for “watch tv”, but we wouldn’t have wanted the same response for “watch sofa”.

Now, we could set up a “weigh” verb for each of our three objects flour, eggs and sugar if we wanted to, but then we would have to enter the same (or a very similar) script for each object. Instead, what we want is *one* script that will let us weigh *any object*. For this, we need to use a **command**. But before we can add a command, we need to understand a bit about **variables**.

11.1 Variables

A variable is something that stores text or a number. For example, a variable called “playername” might store the text “Alex”. A variable called “health” might contain the number “75”.

Usually in Quest, we use variables that store text. In programming, a sequence of text is called a **string**. So, when a variable stores text, it’s called a **string variable**. A string variable could contain the string (i.e. text) “bob” or “look at that” or “Once upon time, in a land far away, there was a cat called Fred.”

Quest also lets you use another type of variable – a **numeric** variable. These contain a number like “3”, “-4.7” or “0” rather than text.

Each variable has a name. You can set the contents of a variable using the script commands in the “Variables” category, and you can read the contents of a variable from any script command by surrounding the variable name with # characters (if it’s a string), or % characters (if it’s a numeric). We will see examples of these later.

11.2 Adding a Simple Command

Let’s add a simple command called “say”. This will let the player type something like “say hello”, and Quest will respond with “You say ‘hello’, but nobody replies.”

Click Commands on the tree menu, and click “Add”. Enter the following text into the command box:

```
say #text#
```

This is what’s called a **command template**. This example command template handles the player typing “say” followed by any text. Whatever text follows the “say” command is put into the string variable called “text”.

For example:

- If the player types “say hello”, the “text” string variable will contain “hello”
- If the player types “say what a lovely day”, the “text” string variable will contain “what a lovely day”

Whenever the player types in a command that matches the command template, the associated script will be run. Let’s now add a script to print the required response, which contains the contents of the string variable called “text”.

To do this, click Edit. The “Print a message” command is already selected for you, so enter the following text into the Message box:

You say “#text#”, but nobody replies.

Click OK. Launch the game and type in a few “say” commands to see that Quest responds correctly.

11.3 Alternative Command Forms

You can easily add alternatives to a command by separating them with semicolons. For example, we could adapt our “say #text#” command to deal with “shout” and “yell” by modifying its command template to read:

say #text#; shout #text#; yell #text#

11.4 Adding a “Weigh” Command

We now know how to add a command that will process any kind of text the player enters. However, a lot of the time, our commands will be dealing with objects that the player can see. When we created verbs earlier, we didn’t have to worry about whether the player used one of the object’s alternative names – our “watch’” verb automatically handles “watch tv”, “watch the tv”, “watch television”, “look at tv” followed by “watch it”, and so on.

Fortunately, a slightly different command template can handle this for us. Let’s add a command now which will let us type “weigh” followed by the name of the object. To do this, add a new command and enter the following command template:

weigh #@object#

Notice the difference? That “@” character tells Quest that it should *only* run this command’s associated script *if* the player has typed in a valid object name. If the player types “weigh car” for example, Quest will automatically respond “I can’t see that anywhere”. This saves us from having to manually get our script to check whether the object name is valid.

So, if our player is in the kitchen and types “weigh flour”, our script will run and the string variable named “object” will contain the text “flour”. If the player types “weigh shoes”, the script *won’t run*, as there is no “shoes” object in the kitchen.

All good so far, but what if the player types “look at flour” and then “weigh it”? Does the “object” variable contain the text “it”? No – Quest handles this for us as well, and it puts “flour” into the string variable.

What if we have several alternative names for our “flour” object and the player uses one of those? What if the “flour” object is in fact called “kitchenobject002” and just has an alias of “flour”? This brings us to an important point – Quest will *always store the code name for the object in the variable*. This means that whatever the player types, you don’t need to do any more processing in your script to work out which specific object the player was referring to.

So:

- If we have several alternative names for the “flour” object, then whichever one the player types, the string variable “object” will *always* contain the text “flour”, since that’s what it’s called internally within QDK.
- If for some reason we’d given the object a name of “kitchenobject002” with an *alias* of “flour”, then when the player typed “weigh flour”, the string variable “object” would contain the code name – “kitchenobject002”.

11.5 Reading the Property

The script we enter for the “weigh” command should respond “It weighs X grams”, where X is the weight of the item – as reported by its “weight” property.

We saw in section 10.2 how to read the custom property of a specific object – we used something like “#eggs:weight#”. But how do we read a property if we don’t know in advance what that object will be?

What we want to do is read the “weight” property of whatever object name is contained within the “object” string variable. Fortunately, this is easy to do by typing:

```
#(variable):property#
```

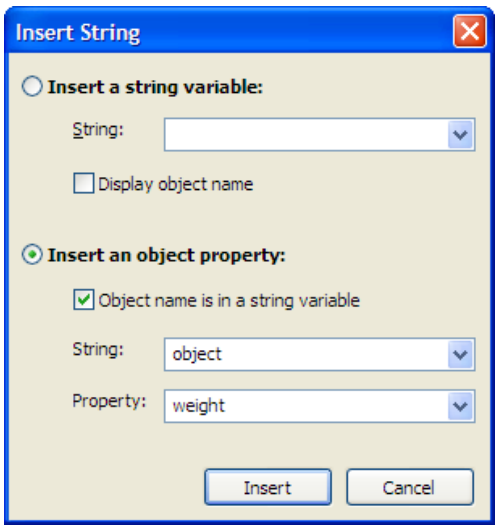
In this case, we need to type:

```
#(object):weight#
```

This will do exactly what we require – it will return the “weight” property of the object specified by the “object” string variable. So if the “object” string variable contains the text “eggs”, the weight of the eggs will be returned. If it contains “flour”, the weight of the flour is returned.

Let’s add this to our script now. With the “weigh” command selected, click the “Edit” button. The “Print a message” command is selected for you, so enter the message “It weighs #(object):weight# grams.”

You can also enter the text “#(object):weight#” by clicking the “#” button and filling in the “Insert String” window like this:



Launch the game and go to the kitchen. See what happens when you type the following:

- weigh car
- weigh flour
- weigh eggs
- look at sugar
- weigh it

Everything should work as you expect. But what happens when you type:

- weigh oven

Quest responds with “It weighs ! grams.” Why? Because the oven doesn’t have a weight property. Since we don’t want to have to enter a weight for every single object in the game, we’ll need to update our command so it checks for the existence of a property, and then prints the appropriate response.

11.6 Checking for a Property

With the “weigh” command selected, click the Edit button to bring up the Script Editor. Click the Cut button to move the existing “Print a message” command to the clipboard. Now click the Add Conditional button.

Click Add to add a condition. Select “An object (or room) has a property”. In the “Object or Room” box, type “#object#”. In the Property box, type “weight”.

By the Then script, click Edit. Click Paste, then click OK. The script we had previously should now be in the “Then” script box.

Edit the Else Script. With the “Print a message” command selected, type the message “You cannot weigh that.”

Click OK to close the Script Editor. Your script should look like this:

```
If "#object#" has the property "weight"  Then Print "It weighs #(object):weight# grams." Else Print "You cannot weigh that."
```

Launch the game and verify that the commands that worked before still give the correct response. Verify that the correct response is given when you type “weigh oven”.

12 “Give” and “Use”, Boolean Properties and Procedures

After a player has taken an object, they can give that object to, or use it on, other objects in the game.

For example, after picking up some flowers, a player can give them to a character called “Susan” by typing “give flowers to susan”.

“Give” and “Use” are set up in exactly the same way – the only difference is whether the player types “give … to …” or “use … on …”.

In this example, we are going to revive the corpse of Bob in the lounge, using a heart defibrillator. First, we need to alter the setup of Bob so that we give a correct description whether he is dead or alive. We could use a flag for this, as we covered in section 9, but instead we are going to use a **Boolean property**.

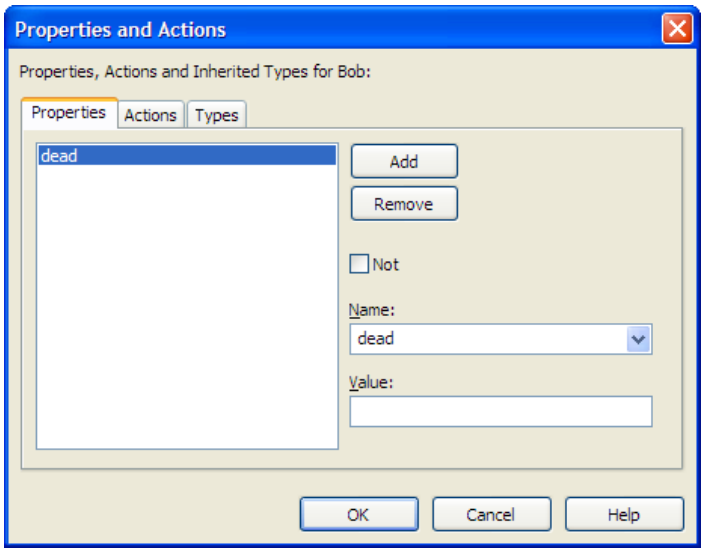
12.1 Boolean Properties

A Boolean property is a type of custom property, like those we covered in section 10. Whereas those properties stored a value, Boolean properties are more like flags – so they are either “on” or “off”.

In this example we will add a Boolean property to Bob called “dead”. If this is on, Bob is dead, and our “look” description will be the same as before. If it is off, Bob has been revived, and we will update our “look” description accordingly.

12.2 Setting the Property

To set the property, select the “Bob” object on the tree menu. Click the Advanced tab and then click the “Edit Properties and Actions” button. Click Add, and enter “dead” in the Name box.



Note that for Boolean properties, we do not enter a value in the Value box. Click OK to close the Properties and Actions window.

12.3 Checking the Property

Now let’s update the “look” description. Go back to the Setup tab and select the “look” verb. Cut the existing description “Bob is lying on the floor, a lot more still than usual.” to the clipboard by selecting it and then either:

- right-click and select Cut

- or press Ctrl+X

Now change the option to “Run a script” and click Edit.

Click the Add Conditional button and click Add to add a condition. Select “An object (or room) has a property”, then select “Bob” from the “Object or Room” list. Type “dead” in the Property box and click OK.

For the Then script, use the “Print a message” command and paste in the old description.

For the Else script, use the “Print a message” command to print “Bob is sitting up, appearing to feel somewhat under the weather.”

The completed script should look like this:

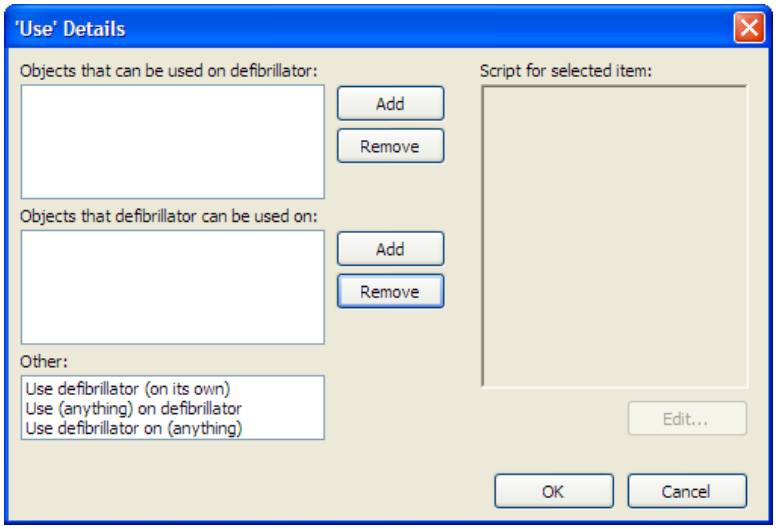
```
If "Bob" has the property "dead"  Then Print "Bob is lying on the floor, a lot more still than usual."
Else Print "Bob is sitting up, appearing to feel somewhat under the weather."
```

12.4 Using a Defibrillator on Bob

Now, add a “defibrillator” object to the lounge. Give it the correct prefix, and enter a description like “A heart defibrillator can magically revive a dead person, if all those hospital dramas are to be believed.”

Go to the Inventory tab and select “Yes” from the “Take” options.

Now, at the bottom of the screen, click the “Edit ‘Use’ Details” button.



This will pop up a window giving you all the various options for using this object:

- Use another object on the defibrillator
- Use the defibrillator on another object
- Use the defibrillator on its own
- Use any object on the defibrillator
- Use the defibrillator on any object

We have several options then, for allowing the player to type “use defibrillator on bob”:

- From this window, we can add “Bob” to the list of objects that the defibrillator can be used on, and then create a script

- From *Bob's* "Use Details" window, we can add "defibrillator" to the list of objects that can be used *on Bob*, and then create a script
- Or we could use one of the "Use defibrillator on (anything)" or "Use (anything) on Bob" options, with a slightly more complicated script.

Since we have the window open already, let's go with the first option. Under "Objects that defibrillator can be used on", click the Add button. Select "Bob" from the drop-down list and click Add.

The Script Editor window will now appear. We want to turn off Bob's "dead" property and print a message. So, go to the Modify category and select Property. Select Bob from the "Object or Room name" list. In the "Property data" box, enter "not dead". This is how we turn off a property – we just put the word "not" in front of it.

Add another script command to print a message saying "Miraculously, the defibrillator lived up to its promise, and Bob is now alive again. He says his head feels kind of fuzzy."

Launch the game, take the defibrillator and then type "use defibrillator on bob". Then look at Bob and verify that the correct description is shown.

12.5 Exercise

Notice that you get the same response if you type "use defibrillator on bob" a second time. Update your script so it prints an appropriate message.

See chapter 27 for the answer.

12.6 Using Procedures

It would be good if we could get the same effect just by typing "use defibrillator". One way we could do this would be to copy the script we just created, and paste it into the "Use defibrillator (on its own)" script. However, if we then wanted to make an update to one script, we would have to update the other one as well.

The best way to resolve this is to make both our existing "use defibrillator on Bob", and our new "use defibrillator (on its own)" *point to the same script*. The way to do this is to set up a **procedure**.

Procedures provide a way for you to set up scripts that can be called from anywhere in your game, so you don't have to keep copying and pasting or re-entering the script.

Let's create one now to store the script commands we use to resuscitate Bob. We can then set both "use defibrillator on bob" and "use defibrillator (on its own)" to call this procedure.

First, select the "use defibrillator on bob" script and click Edit. Click the Cut button to move the existing script to the clipboard, and then click OK. Click OK to close the "Use Details" window.

Now select Procedures from the tree menu on the left side of the screen (in the "Game" section, underneath Verbs and Commands). Click the Add button to add a new procedure named "revive bob". Click the Paste button to paste the script into this procedure, and then click OK.

Now we just need to update "use defibrillator on bob" and "use defibrillator (on its own)" to call this procedure. Go back to "use defibrillator on bob" and click Edit to create the script. Go to the "Run script" category and select the "Run a procedure" command. From the drop-down Procedure list, select the "revive bob" procedure which we just created. Click OK.

The “use defibrillator on bob” script is now:

```
Run the "revive bob" procedure
```

The “revive bob” procedure contains the script that was in there before, so the effect of the player typing “use defibrillator on bob” is unchanged.

Select “use defibrillator (on its own)” and edit its script so that it also calls the “revive bob” procedure.

Launch the game now and verify you get the same response whether you type “use defibrillator on bob” or just “use defibrillator”.

Note that you can also type “use defibrillator” if you’re in the kitchen, and it still works – which would be remarkable to say the least. Later on we’ll see a way of checking what room the player is in before running a script (but if you just can’t wait that long, you can take a look at section 16.3).

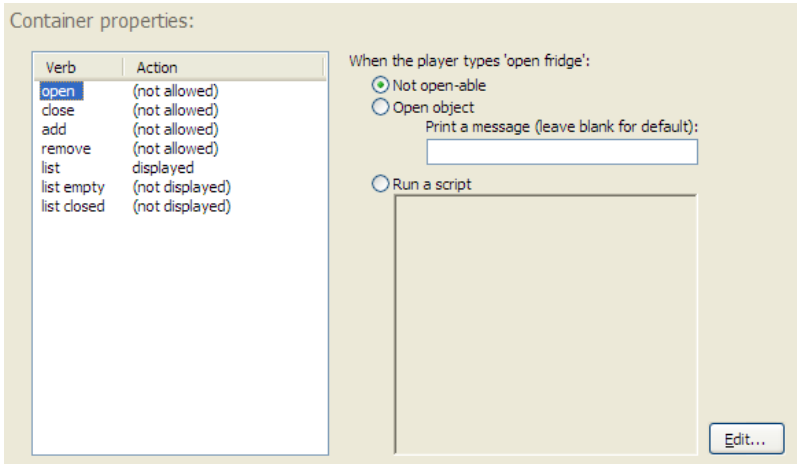
13 Adding a Container – the Fridge

Containers are objects which can contain other objects. In this example, we'll create a "fridge" object in the kitchen, which contains several items of food and drink. The fridge is initially closed, so these items will only be visible once the player has opened the fridge.

13.1 Creating the Fridge

Create a "fridge" object in the kitchen and give it the prefix "a". Give it a description of "A big old refrigerator sits in the corner, humming quietly."

Now let's set the fridge up as a container. Click the Container tab. By default, the "Not a container" option is selected. Change this to "Container". The Container Properties section of the window will now be enabled.



The following default options will be set:

- **open:** The object cannot be opened
- **close:** The object cannot be closed
- **add:** Objects cannot be added by the player
- **remove:** Objects cannot be removed by the player
- **list:** Any objects inside the container will be listed after the object description
- **list empty:** No additional description is printed when the object is empty
- **list closed:** No additional description is printed when the object is closed

These are pretty restrictive. Let's set the fridge up so the player can open and close it. To do this, click "open" from the list of container verbs. Underneath "When the player types 'open fridge'", select "Open object". This makes the fridge open-able.

Now follow a similar procedure to make the fridge close-able too.

13.2 Adding Objects to the Fridge

Now let's create some objects inside the fridge. To do this, we just create these as normal objects inside the kitchen. The only difference is that we set the object's **parent** to "fridge", to tell Quest that these objects are inside the fridge.

Add the following objects:

- milk

- cheese
- beer

Give each object a sensible prefix and description, and set their parent to “fridge”. Allow each object to be taken.

13.3 Running the Game

Now run the game and go to the kitchen. Notice that you can’t see the milk, and if you type something like “look at milk”, Quest will tell you that it’s not here. Now open the fridge, and the objects inside it will be revealed.

13.4 Adding and Removing Objects During the Game

It would be a good idea if players could also put things inside the fridge. To do this, go back to the fridge’s container verbs and select “add”. Select the “Add object” option to allow players to add things to the fridge. Likewise, allow players to remove objects from the fridge by setting the appropriate option from the “remove” verb. Run the game and you should now be able to put the flour, eggs and sugar objects we created earlier into the fridge.

Note that:

- Objects can only be put in a container (or on a surface) if they can be taken.
- When the player tries to take an object that is in a container, that container must allow objects to be removed from it, and the object itself must also be takeable.

13.5 Running Scripts

Instead of automatically allowing the player to open, close, add or remove objects from a container, you can run a script. This can be useful if, for example, the container is locked – then you can only allow the player to open the container if they have the key or know the correct combination. To add a script, just select the “Run a script” option. You can then open or close the object by using the commands in the “Objects” category.

To add an object, you’ll need to know what object the player typed. Quest stores the object name in the #quest.add.object.name# and #quest.remove.object.name# string variables as appropriate, so you can use these as the object names in the “Add an object to another object” and “Remove an object from its parent object” commands.

13.6 Checking if an Object is Opened – Updating the Description

To check whether an object is opened or not, you just need to check whether the “opened” property of the object is set.

In this example, we’ll update the description of the fridge according to whether it is opened or closed.

Select the fridge and copy the existing description “A big old refrigerator sits in the corner, humming quietly.” to the clipboard. Change the “look” option to “Run a script” and click Edit.

Click Add Conditional and then add the condition “An object (or room) has a property”. Select the fridge object and enter the property “opened”. Click OK.

In the Then script, print the message “The fridge is open, casting its light out into the gloomy kitchen”. In the Else script, print the old description “A big old refrigerator sits in the corner, humming quietly”.

Your completed script should look like this:

```
If "fridge" has the property "opened"  Then Print "The fridge is open, casting its light out into the gloomy kitchen." Else Print "A big old refrigerator sits in the corner, humming quietly."
```

Now run the game and look at the fridge. Open it, and then look at it again to verify that the description is updated correctly.

13.7 Listing the Contents

By default, when a player looks at an object that has contents, such as our fridge, Quest will display what’s inside it at the end of the “look” description. We can control this by setting the relevant option under the “list” verb.

The simplest thing we can do is change the header for the list. This defaults to “It contains”, followed by the list of objects. If you put a colon at the end of the header, the list of objects will be printed. If not, the only thing that will be printed is what we type in the box.

For maximum flexibility, we can also run a script to print a custom description.

13.8 When the Fridge is Closed or Empty

By default, when the object is closed or empty, Quest won’t print any additional description after “look” because there won’t be any available objects to talk about. If you want though, you can get Quest to print “It is closed” or “It is empty” by selecting the relevant option from the “list empty” and “list closed” verbs.

13.9 Exercise

As an exercise, add a cupboard to the kitchen which is initially closed. Add a few items to the cupboard such as a tin of beans, a packet of rice etc. The player should be able to open and close the cupboard. When Quest lists the contents of the cupboard, it should say something like “The cupboard is bare except for …”

13.10 Transparency

When you set the “Transparent” option, the player can see what objects are inside the container, even if it is closed. This means the “list closed” verb is not applicable to transparent objects.

Although the player can see what’s inside a transparent container, they still can’t take objects from it or add objects to it unless it is open.

13.11 Surfaces

Surfaces act very much like containers, although they are rather simpler. Because a surface can’t be opened or closed, the “open”, “close” and “list closed” verbs are not applicable to surfaces. A surface acts like an always-open container, and objects that are on a surface are visible in a room description even before the player has looked

at the surface. For this reason they're a good choice for implementing things like tables.

As an exercise, change the table object in the lounge to make it a surface, and set up the newspaper so that it is on the table.

14 Bringing Objects into Play during the Game

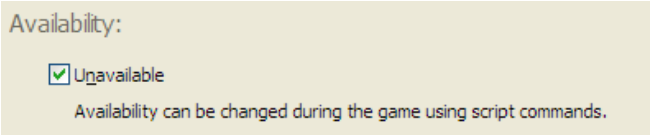
As your game unfolds and the player interacts with your world, you may want to bring additional objects into play, or remove others. In this example, we'll add a window to the kitchen. When the player opens it, a bee flies in. In the next chapter we'll make this bee quite irritating.

14.1 Creating a Hidden Object

First, let's create the bee object. In the kitchen, add an object called "bee". Give it a prefix of "a" and a suitable description.

When the game begins, we don't want the bee object to appear. It will only appear when the player has opened the window. To do this, we set the object to be initially **unavailable**.

Click the Advanced tab, and tick the "Unavailable" box.



Launch the game and go to the kitchen. You will see that the kitchen appears exactly as it did before – the bee object cannot be seen, and if you type "look at bee", you will get the same response as if you type "look at wolf" – although we have defined the object in the kitchen, because it is unavailable Quest acts as if it wasn't there at all.

14.2 Bringing the Object into Play

Now, add a window object to the kitchen and give it a sensible prefix and description. We want to make this window openable, so let's set it up as a container (OK, so a window doesn't really contain things, but that doesn't matter – since we won't be adding anything to it, and players won't be able to either, by default). Go to the Containers tab and select "Container". Now go to the "open" verb and choose the "Run a script" option. Click Edit.

From the Objects category, choose the "Make an object accessible" command. Select "bee" from the objects list. Then add a "Print a message" command to print "You open the window and a bee flies into the kitchen."

Launch the game and go to the kitchen. Open the window and verify that you can now look at the bee.

14.3 Checking if the Object is Already There

What if the player types "open window" a second time? At the moment they'll get told that the bee has flown in again, which doesn't make sense as it is already there. There are several ways you can get round this – we have already looked at flags, for example. We could also check the "opened" property of the window. However there is another way we can do this – we can check whether the bee has already appeared. If it has, the player has already opened the window, so we just need to print "The window is already open."

With the window object selected, select the "open" container verb and edit the script. Select both the existing script lines (you can hold down the Control key to select multiple script lines) and click the Cut button.

Click “Add Conditional” and then add a condition. Select “An object is available for interaction” and enter “bee” in the object box. Click OK.

By the Then script, click Edit. With “Print a message” selected, enter “The window is already open.”

Edit the Else script. Click Paste and then click OK.

The completed script should look like this:

```
If "bee" is available for interaction  Then Print "The window is already open." Else {
Make "bee" accessible
Print "You open the window and a bee flies into the kitchen." }
```

Launch the game and verify that if you open the window twice, you get a sensible response.

14.4 Removing an Object during Play

As well as bringing an object into play, you can also remove an object from play using the “Make an object inaccessible” command from the Objects category.

As an exercise, add an “apple” object, with a sensible prefix and description. Add an “eat” verb to the object which will print a message saying “You eat the apple. Tasty.” and then remove the apple from play.

15 Exercises

If you've got this far, well done – you're well on your way to knowing everything you need to create your first text adventure game. In fact, we've covered the vast majority of things that most games need. The remaining chapters cover a number of more advanced topics, so this is a good point to check whether you've understood the information in the previous chapters.

In this exercise, I want you to add a tap in the kitchen (if you're American, you may prefer to call it a faucet). You should also add a glass. Here's what should happen:

- The player turns on the tap/faucet to start the water flowing.
- The player picks up the glass.
- The player fills the glass. (This can only happen if they have taken the glass)
- The player drinks the water. (This can only happen if they are holding a glass full of water).

At each stage:

- Quest should give a suitable response for "look at glass" and "look at tap".
- If the player tries to do something that they are not allowed to do, for example they try to drink the water when they have not filled the glass, Quest should give a suitable response.
- The player should be able to perform the above procedure using any reasonable commands.

If you get stuck, take a look through the previous chapters and see if you can work out how this should be implemented. Once you've worked it out, go ahead and create the necessary scripts and anything else you need.

Tip: Although in real life the glass would act as a "container" for water, you don't actually need to use containers to create this example.

When you have finished, or if you just really really can't help cheating, turn to chapter 27 for one way of implementing this.

16 Using Timers

In chapter 14, we made a bee fly into the kitchen after the player opened a window. We'll now make that bee a bit more annoying as it flies around the kitchen – every 20 seconds, it will buzz past the player.

To do this, we will use a timer. This timer will only be activated when the player opens the window in the kitchen. When the timer is activated, every 20 seconds it will print the message "The bee buzzes past you. Pesky bee." This message will only be printed if the player is in the kitchen.

First, let's set up the timer. After we have done this, we will add the script command to activate it at the right time.

16.1 Setting up the Timer

Select Timers from the tree menu. The Timers screen will open, giving you options to Add, Remove and Edit the timers. We don't have any timers set up yet, so let's add one by clicking the Add button.

Enter the name "bee timer".

The Edit Timer window will now open. The Interval specifies how often the timer fires – in this case, we want it to fire every 20 seconds, so enter "20". Leave the "Initially timer is on" box unticked.

Click the Edit button to edit the script to run when this timer fires. With "Print a message" selected, enter the message "The bee buzzes past you. Pesky bee." Click OK.

Click OK to close the Edit Timer window.

16.2 Activating the Timer

Go back to the "window" object and edit the "open" container verb script. Edit the Else script and click the "Add Command" button. Select the Timers category and choose the "Turn on a timer" command. From the drop-down "Timer name" list, select "bee timer" and click OK.

The script should now look like this:

```
If "bee" is available for interaction  Then Print "The window is already open." Else {
Make "bee" accessible
Print "You open the window and a bee flies into the kitchen."
Turn on timer "bee timer" }
```

Launch the game, go to the kitchen and open the window. Wait for a while and verify that the message is printed every 20 seconds.

Now go north to the lounge. You'll see that we still get the message about the bee flying around. Woops! That bee is only in the kitchen. We'll need to update the timer script so that it only prints the message if the player is in the kitchen.

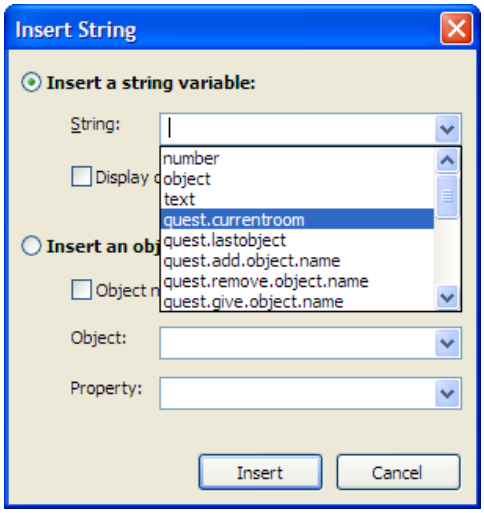
16.3 Checking where the Player is

Go back to the Timers screen and edit the "bee timer". Click the Edit button to edit the script.

Click Cut to copy the existing script to the clipboard. Click "Add Conditional" and then click "Add" to add a condition.

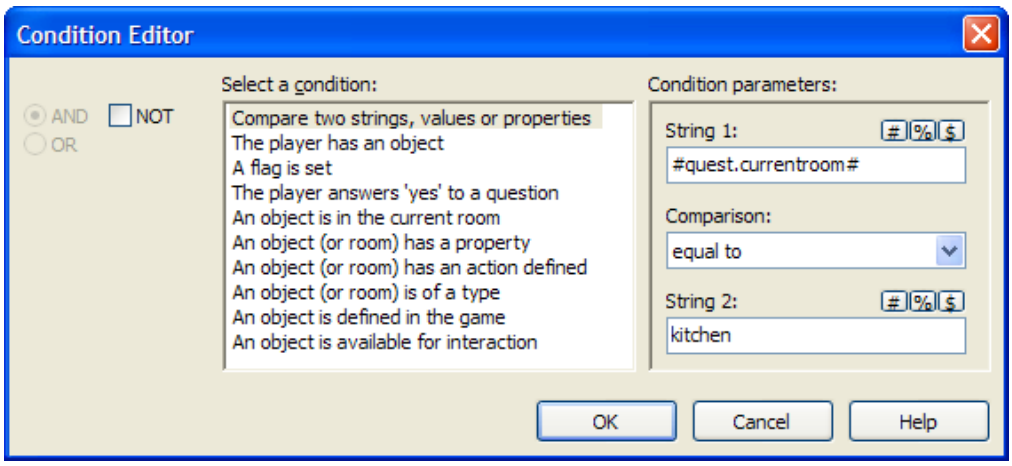
We're now going to add a condition to check if the player is in the kitchen. Quest stores the player's current room in a string variable called "quest.currentroom".

With the "Compare two strings, values or properties" condition selected, click the "#" button next to "String 1".



From the drop-down list, select "quest.currentroom" and click Insert. The text "#quest.currentroom#" will be inserted into the "String 1" box.

In the "String 2" box, type "kitchen".



Click OK.

By the "Then" script, click Edit. Click Paste and then click OK.

Click OK to close the Script Editor. The finished script should look like this:

If "#quest.currentroom#" is equal to "kitchen" Then Print "The bee buzzes past you. Pesky bee."

Click the OK buttons to confirm your changes to the “bee timer”.

Launch the game, go to the kitchen and open the window. Verify that the message still prints while we are in the kitchen. Go north to the lounge and verify that the message doesn’t print while we are there.

17 Creating a Turn Counter

In this chapter, we'll take a look at a number of concepts:

- scripts that run **after each turn** in the game
- **numeric variables**, allowing us to manipulate numbers
- **status variables**, allowing us to display information permanently on-screen

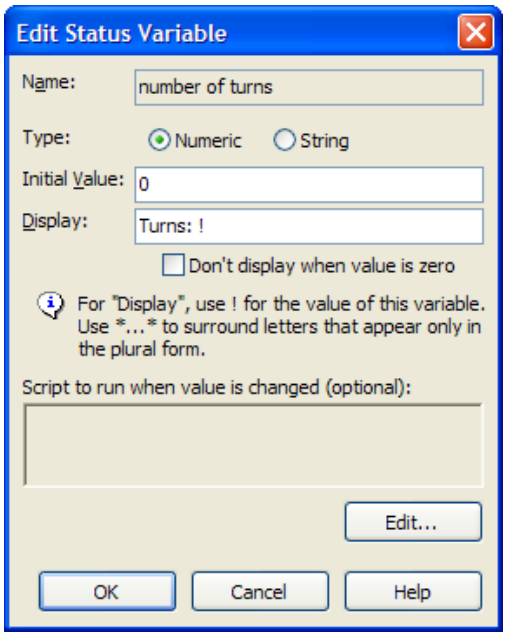
We'll use all of these to create an on-screen turn counter.

Here's how we're going to do it:

- we'll store the number of turns a player has taken in a numeric variable called "number of turns"
- this variable will be set up as a status variable, so it is always displayed on-screen
- we'll create an "after each turn" script which will increase the number stored in the "number of turns" variable
- so, every time the player takes a turn, the number is increased and the display is automatically updated

17.1 Setting up the Status Variable

To set up our "number of turns" variable, select Status Variables from the tree menu. Click Add and enter the name "number of turns". The Status Variable window will open, with the type set to "numeric" by default.



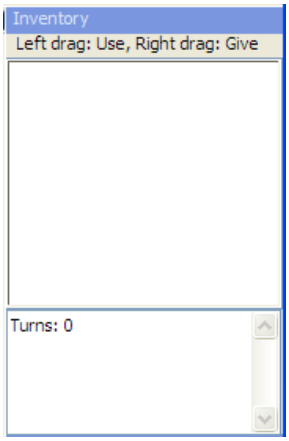
Enter "0" as the initial value.

The "Display" box lets us specify how the variable should be displayed. It will appear in a box underneath the Inventory on the right-hand side of the Quest window. We want it to say "Turns: 0", "Turns: 1", "Turns: 2", and so on. To do this, type "Turns: !" in the "Display" box. Quest will replace the exclamation mark "!" character with the value of the variable.

Leave "Script to run when value is changed" blank.

Click OK to close the Edit Status Variable window.

If you launch the game now, you should see the status variable displayed on the right-hand side of the Quest window. We've not yet added the script to increase the value of this though, so it will always say "Turns: 0" no matter how many turns we take. Let's add this script now.



17.2 Increasing the Turn Counter after Each Turn

Click “Game” on the tree menu at the left of the QDK window, and select the “Advanced” tab. Under the “Advanced script commands” section, select the “After each turn” script and click the Edit button.

We’re going to add a script command which will increase the value in the “number of turns” variable by 1 each time it is called. Go to the Variables category, and choose the “Increment a numeric variable” command.

In the “Numeric variable name” box, enter “number of turns”.

You can leave the “Increment amount” box blank, as by default Quest will increment our numeric variable by 1.

Click OK to close the Script Editor. That’s all there is to it – after each turn the player takes, the “number of turns” value will be increased, and this new value will be updated on the window. Launch the game now and see that whenever you type a command, the “Turns” display is automatically updated.

17.3 Displaying Plural Values

Our turn counter currently says “Turns: 0” and so on, but what if we wanted it to say “You have taken 0 turns”? Go back to the Status Variables window and edit the “number of turns” variable. Change the “Display” box so it reads “You have taken ! turns”.

Now launch the game again and take a few turns. You will see that this pretty much works, apart from after we’ve taken one turn and it reads “You have taken 1 turns”. This doesn’t sound quite right.

The way around this is to let Quest know that the final “s” only appears in the plural form. To do this, you just surround it with asterisks “*”. Go back to edit the “number of turns” variable again, and this time change the “Display” box so it reads “You have taken ! turn*s*”.

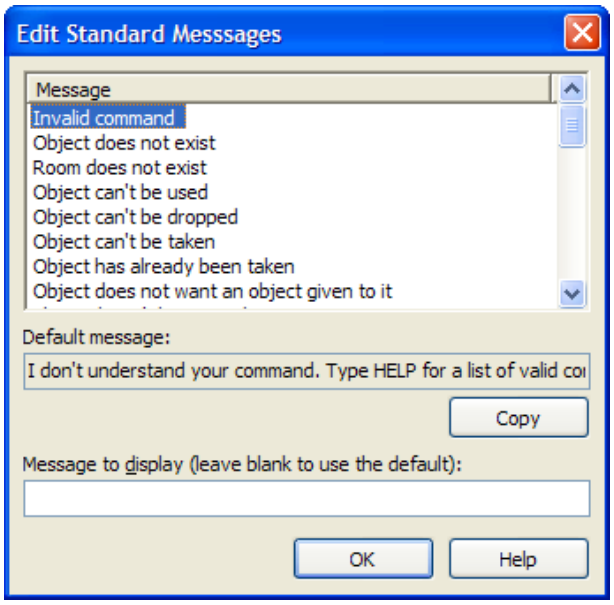
Launch the game again and verify that the turn counter now makes grammatical sense.

18 Changing the Standard Responses

When a player types in a command that Quest doesn't recognise, it will by default respond "I don't understand your command. Type HELP for a list of valid commands." This is all very well, but it might break the flow of our game if it doesn't fit in with the rest of the text that we've written. Fortunately, Quest provides a way for you to change all of its default responses.

18.1 Editing the Standard Messages

Click the Tools menu and select "Standard Messages". The Edit Standard Messages window appears. This lists all of Quest's default responses and allows you to edit them.



Let's edit that "I don't understand your command. Type HELP for a list of valid commands" message. Select the first message in the list, "Invalid command".

You will see the default text is displayed. If you click the Copy button, this text will be copied into the Message box below, allowing you to edit it. Alternatively, you can type your own message.

Enter the message "I don't know what you're talking about. Type HELP to see what kinds of commands you can use."

Launch the game and type some nonsense into the command box, to verify that your custom message is displayed instead of Quest's default.

As an exercise, find the message "Nothing out of the ordinary" that prints as the default "look" description. Change it to read "It's not very exciting."

18.2 Modifying the Standard Verb Responses

Over the course of this tutorial, we've added several verbs to our game. One of the first verbs we added was "watch", which we set up so that the player could type "watch tv". If the player tries to watch anything else, Quest responds "You can't watch that."

You can change this text – in fact you can run any script when the player uses a verb that has not been explicitly defined for an object. Let's change our default "watch" text to "You stare at it for a while, but nothing much happens."

To do this, select “Verbs” from the tree menu on the left-hand side of the QDK window, and edit the “watch” verb. The script defaulted to

```
Print "You can't watch that."
```

when we first set up the verb. Click the Edit button and modify the message to “You stare at it for a while, but nothing much happens.”

Launch the game and verify that the correct response is given when you type something like “watch sofa”.

What if you type in “watch bob”? The same response is given, but it would be more polite if Quest properly used the object’s article (“him”) rather than just “it”. To do this, we need to read the article property of whatever object the player has typed in.

If I tell you that Quest stores the name of the object that the player uses in the string variable “quest.lastobject”, you should be able to work out how to access the “article” property of the object, using the knowledge you gained in section 11.5.

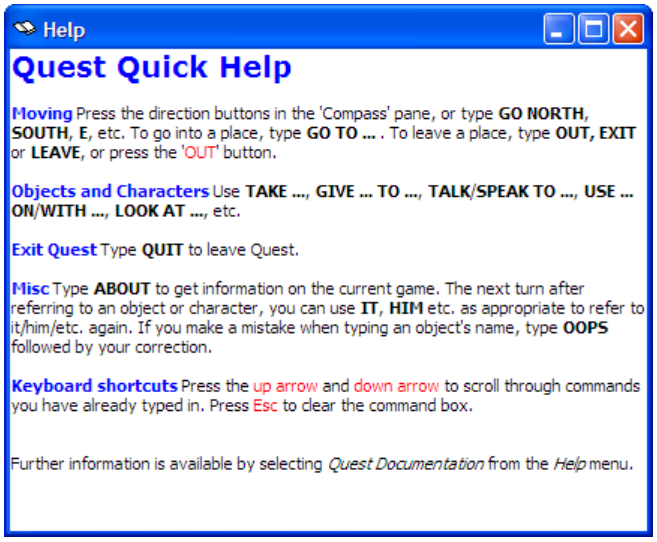
You may want to take another look at that section now. If not, read on for the answer.

Edit the default “watch” script again and change the message to “You stare at #(quest.lastobject):article# for a while, but nothing much happens.”

Launch the game and verify that Quest gives you a sensible response for “watch newspaper” and “watch bob”.

18.3 Creating your own Help Text

When a player types “help”, a pop-up window appears with the Quest Quick Help guide, listing the standard commands a player can use.



You may want to customise this so that it fits in better with your game.

To do this, you need to override the “help” command so that instead of Quest taking its default action of showing the help window, it runs whatever script you specify instead. All you need to do is add a “help” command to your game, as this will automatically take precedence over Quest’s built-in command. (You can in fact override all of Quest’s built in commands this way – even things like “west” or “take”, if you’re feeling adventurous).

Click “Commands” on the tree menu and click Add to add a command. Enter “help” as the command template. Click Edit to bring up the Script Editor.

You can type whatever message you want to appear when the player types “help”. If you want to pop up your own help window, you can use the commands in the “Help

window” category. As an example, choose the “Print a message in the help window” category and type “Here is some example text in the help window.”

Launch the game and type “help” to verify that this works.

We won’t worry about creating our own help guide for the moment, so if you wish you can delete the example “help” command you just created.

19 Text Formatting and Text Blocks

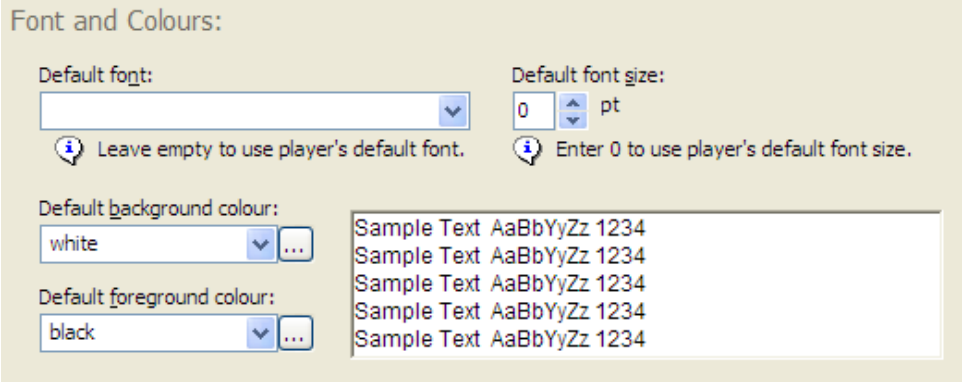
Up until now, our Quest game has been entirely plain text. In this section, we'll look at jazzing things up a bit by using text formatting and including pictures. We'll also look at an easy way to include large sections of text, by using text blocks.

19.1 A Note about Text Formatting

Bear in mind when using text formatting that not all players will be able to see it. Some people may have turned off text formatting entirely – either due to visual impairments, or because they simply don't like it. For this reason, you should always ensure that your text formatting is not essential to the game working correctly.

19.2 Setting the Default Font and Colours

To change the default font and colours used throughout your game, select "Game" from the tree menu and go to the Display tab.



If you leave the Font name option blank, your game will use Quest's default font. Usually this is set to Arial, though the player may have changed this to something else.

If you leave the Font size option set to 0, your game will use Quest's default font size – usually 9pt, though again the player may have set this to something else.

To change the default foreground and background colours, select the relevant option from the drop-down lists, or click the buttons to choose from a colour palette.

Let's experiment with this now:

- Change the font to "Lucida Console"
- Change font size to 12pt
- Change the background colour to blue
- Change the foreground colour to white

Launch the game and marvel at the retro look you have created.

Now change the formatting back to how it was before.

19.3 Changing Fonts and Colours during the Game

You can also change the font and colours while the game is in progress. You can do this from any script command. The commands are all in the "Print" category.

For our example, we'll set the background colour to yellow while the player is in the kitchen, and to white when the player is in the lounge.

Select the kitchen room from the tree menu and click the Advanced tab. Click the "Edit" button to edit the "When the player enters the room" script.

Select the "Change the background colour" command from the "Print" category, and select "yellow" from the drop-down box.

In the same way, edit the lounge room so that the background is changed to white when the player enters.

Now launch the game and verify that the background colour changes as you move between rooms.

There's not much point to this of course, so you may now want to remove the commands we've just added. The players of your game would soon get tired of such gimmicks.

19.4 Inserting Pictures

You can show a picture using the script commands in the "Pictures & Sounds" category. The "Show a picture" command will show the picture in the main Quest window, but you can use the "Show a picture in a popup window" command instead if you'd like it to be displayed separately.

In this example, we'll display a picture when the player enters the kitchen.

First, find a picture of a kitchen.

- You could use Google Images (http://images.google.com), which will be fine for our demonstration – but bear in mind if you want to distribute images with your game, they must be images you've created yourself, or at least you must have permission from the copyright owner.
- A good website for royalty free images is www.istockphoto.com.
- Or you could just go to your own kitchen with a digital camera.
- Or, since this is just a demonstration, you could find a picture of anything else at all.

Now, in QDK, click the "kitchen" room on the tree menu. Click the Advanced tab, and click the "Edit" button for the "When the player enters the room" script. Go to the "Pictures & Sounds" category and select the "Show a picture" command. Click Browse and select your picture file from where you saved it. Click OK. Your picture will be automatically copied to the same folder as your game's ASL file.

Now launch the game and go to the kitchen to verify that the picture is displayed with the room description.

19.5 Using Text Blocks

We looked briefly at text blocks in section 3.2, where we edited the "intro" text block. You can also add as many text blocks of your own as you like, which can be useful if you have a large amount of text to write.

To do this, click Tools menu and select Text Blocks. Click the Add button and enter a name for your text block. Enter some text and click the Close button.

Once this is set up, you can show this text block from any point in the game by using a script command. You could, for example, create a text block to contain the description of a room. You could then show the description of the room from a "Run script" style description, using the "Display a text block" command from the "Print" category.

20 Select Case – Dialling a Telephone

A “select case” is like a conditional script, but it caters for more options. In this example, we’ll add a telephone to the lounge and set up a “dial” command. The player will be able to dial a variety of different numbers.

- 911 or 999 will print the message “There's no need to call the police now.”
- 94672 will print the message “Madame Buxom, Queen of Pain, is on her way.”
- 32461 will print the message “A voice at the end of the line says ‘Stop calling this number you pervert!’”
- 15629 will print the message “You’re not hungry or drunk enough for anything from Luigi’s Pizza Palace right now.”
- Otherwise “Sorry, wrong number!” will be printed

We could use a normal conditional script for this, but if we had a lot of potential numbers the player might dial, we would quickly end up a lot of “Else” scripts containing more script – it would look like this:

```
If "#number#" is equal to "911"  or "#number#" is equal to "999"  Then Print "There's no need to call the police now." Else If "#number#" is equal to "94672"  Then Print "Madame Buxom, Queen of Pain, is on her way." Else If "#number#" is equal to "32461"  Then Print "A voice at the end of the line says ‘Stop calling this number you pervert!’" Else If "#number#" is equal to "15629"  Then Print "You’re not hungry or drunk enough for anything from Luigi’s Pizza Palace right now." Else Print "Sorry, wrong number!"
```

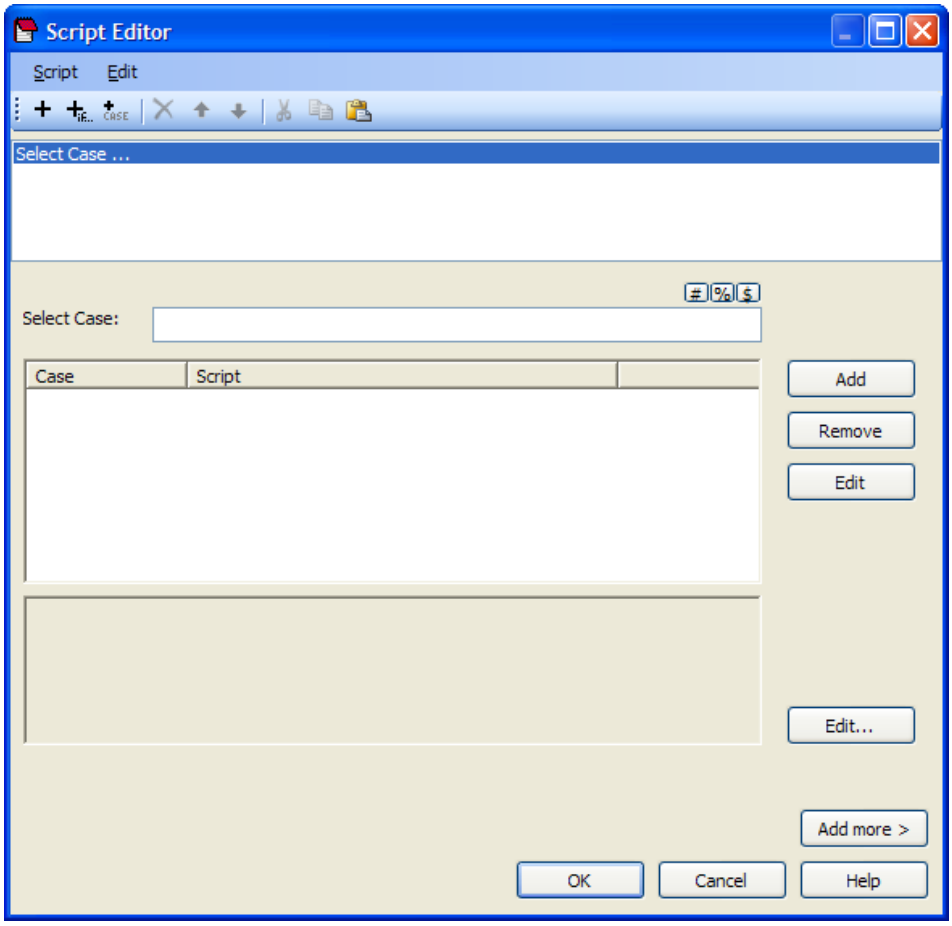
If you try and create this script, you will see it’s quite a pain – far too much clicking and entering the same thing over and over again. The way to do this using the least effort is to use a **select case**.

20.1 Creating the Select Case Script

First, add a telephone to the lounge and give it a sensible prefix and description. Next, add a command “dial #number#”. Whenever the player types “dial”, the text that follows will be put into the “number” string variable. Rather than using lots of conditional scripts, we will use just one select case to print the correct response.

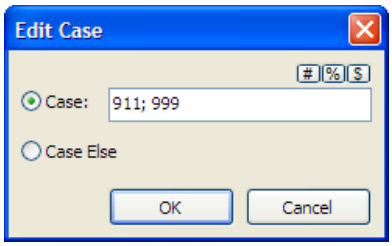
Click Edit to edit the script for the “dial #number#” command. Click the “Add Select Case” button  to create the select case script.

The select case editor will be displayed:



In the “Select Case” box, enter “#number#”. This is what we’re using as the basis for the comparison – each of the **cases** that we add will be compared against this.

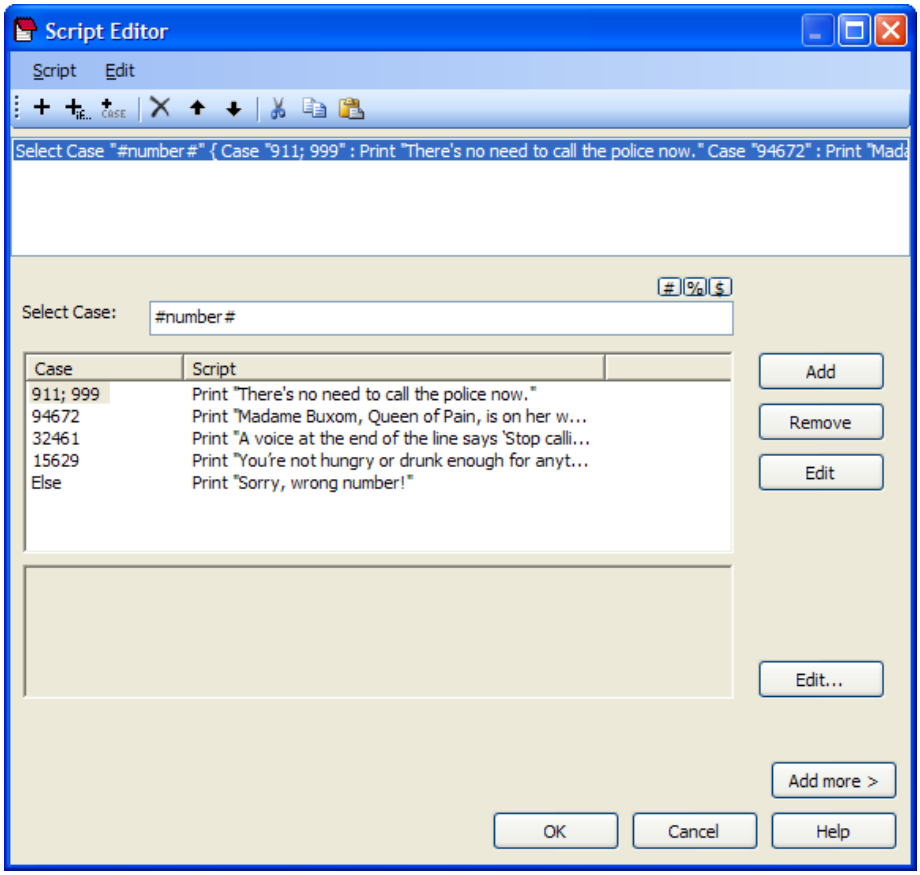
Click Add to add the first case. The Edit Case window will appear. Enter the case “911; 999”.



Click OK. The Script Editor will now be displayed, ready for you to enter the script that should run if #number# is equal to 911 or 999. With “Print a message” selected, enter the message “There's no need to call the police now.” and click OK.

Follow the same process to enter the responses for the other phone numbers. When you reach the bottom of the list, select the “Case Else” option to print “Sorry, wrong number!”.

When you have finished, the Script Editor should look like this:



Click OK and see the script is entered like this:

```
Select Case "#number#" {
Case "911; 999" : Print "There's no need to call the police now."
Case "94672" : Print "Madame Buxom, Queen of Pain, is on her way."
Case "32461" : Print "A voice at the end of the line says 'Stop calling this number you pervert!'"
Case "15629" : Print "You're not hungry or drunk enough for anything from Luigi's Pizza Palace right now."
Case Else : Print "Sorry, wrong number!"
}
```

This is much easier to read, edit and add to than if we had used lots of conditional scripts.

Launch the game and dial a few numbers to check that you see the correct response.

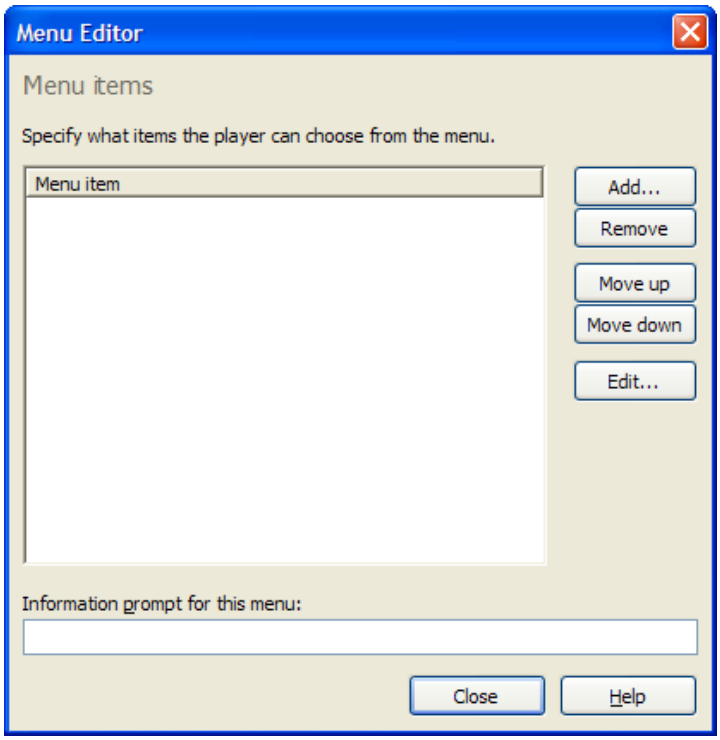
21 Selection Menus – Adding Stories to the Newspaper

Selection menus allow you to give the user a choice between several items. In this example, we're going to update the newspaper's "read" script to give the player a choice of stories to look at.

21.1 Creating the Menu

First, let's create the menu. To do this, click Menus from the tree menu. You will then see the Menus screen, which gives you the ability to add, remove and edit menus in this game. There aren't any menus yet, so click Add to create one. Enter the name "newspaper articles".

You will now see the menu editor screen.

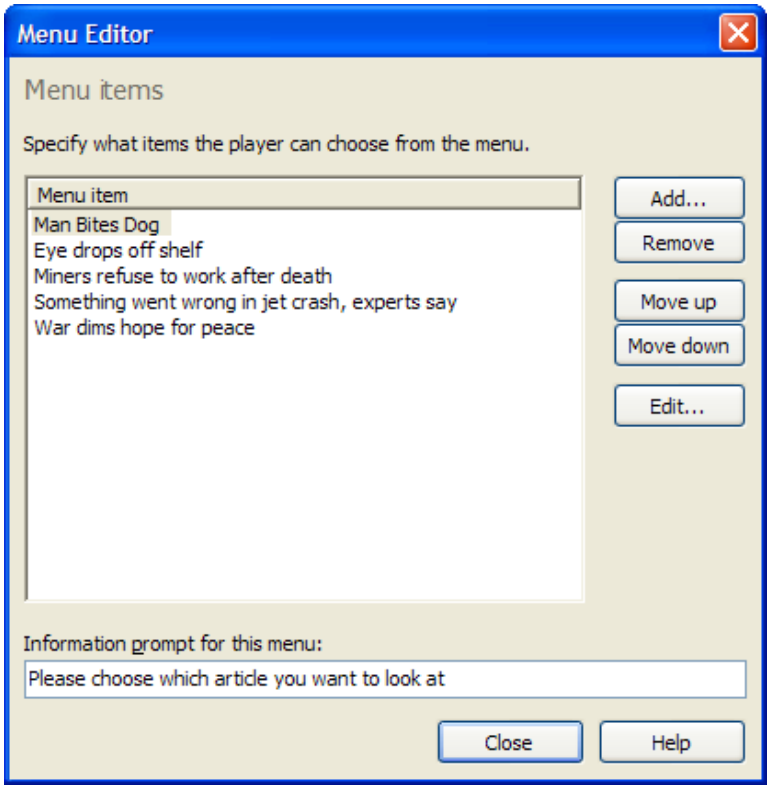


The information prompt is the question that we're going to ask the player. Enter "Please choose which article you want to look at".

You can now enter as many choices as you like. Each choice has a particular script. When the player chooses that particular choice, the script for it is run.

Let's enter our first menu choice. Click "Add" and enter a headline – "Man Bites Dog" for example. The Script Editor will then be displayed. Print a message with the contents of the newspaper article – since this is just a demonstration you don't need to write anything amazingly long or witty, unless you really want to. Something like "A local man was arrested today for biting his pet dog. "I was just really hungry," he said." will do for now.

In the same way, add a couple more headlines and stories. When you have finished, the Menu Editor should look something like this:



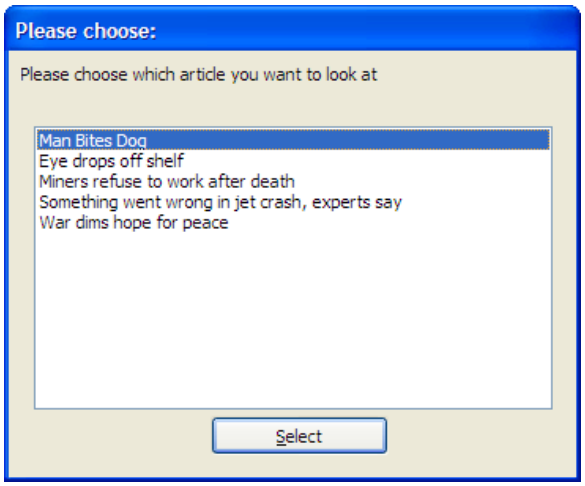
21.2 Calling the Menu

Now all we need to do is make this menu appear when the player reads the newspaper. Select the newspaper object, and change the “read” verb to the “Run a script” option. Edit the script and select the “Run script” category, then choose the “Show a menu” command. From the “Menu name” list, select the “newspaper articles” menu we just created.

The script should look like this:

```
Show the "newspaper articles" menu
```

Now run the game and read the newspaper. You will see the menu appear:



When you select an option, the text you entered for it should be displayed.

22 Additional Topics

In this chapter, we'll take a brief look at the other features of QDK which we've not yet covered.

22.1 Adding a Window Menu

You can add a menu to your Quest game to sit alongside the Quest, Debug and Help menus at the top. Each menu item in your menu will cause a particular command to be run, just as if they player had typed it in.

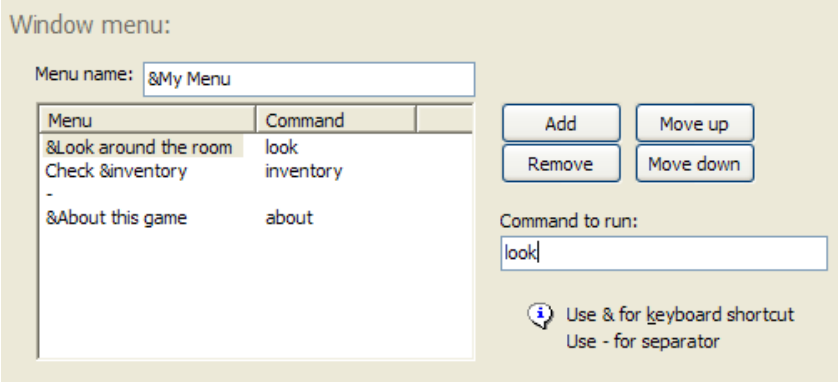
To add a menu, click Game on the tree menu and select the Display tab. Enter a title for the menu and then add the menu items you need, together with what command should run when the player clicks that menu.

For example, you could add a "Look around the room" menu item to call the "look" command.

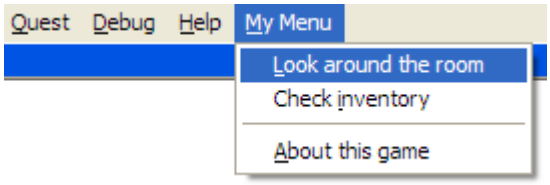
To add a keyboard shortcut to a menu item, enter an ampersand character "&" before the letter to use.

To insert a separator in the menu, enter a dash "-".

Below is an example menu:



This appears in Quest like this:

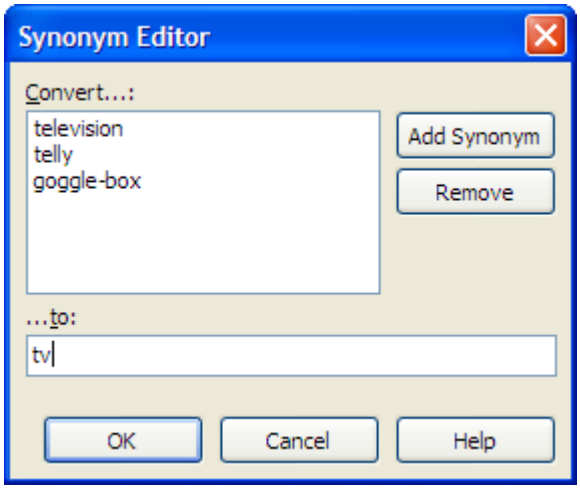


22.2 Synonyms

Synonyms give you another way of setting up alternative words for things. For example, you could set up synonyms so that whenever the player types "television", "telly" or "goggle-box", it is translated internally to "tv". Or you could set up synonyms so that whenever the player types "shout" or "yell", it is translated internally to "say".

Usually a better way of doing this is to set up alternative names for an object, or alternative forms for a command or verb. However, the flexibility is there if you wish. To set up synonyms, click select Synonyms from the tree menu.

To add our "tv" example, click Add. Click the "Add Synonym" button to add "television", "telly" and "goggle-box" to the list, and enter "tv" in the "to:" box.



22.3 Functions

Functions are like procedures – they are bits of script that can be called from anywhere. Unlike procedures, they return a value by using the “Set this function’s return value” command from the “Advanced” category.

You can call a function from within any script command, in a similar way to using a string or numeric variable. All you need to do is surround the function name in $ characters.

For example, say you have a “Print a message” script which prints the message:

> “You have $time$ left to live”

The “time” function will be called, and whatever value it returns will be put into the message. So if the “time” function returns “2 hours”, the player will see the message “You have 2 hours left to live”.

22.4 Built-in Functions

As well as creating your own functions, Quest has a number of built-in functions which allow you to do useful things such as processing text and generating random numbers.

To find out about Quest’s built-in functions, see the “Built-in Functions” section in the ASL Reference guide in the Quest documentation.

22.5 Actions

We’ve already covered object properties in some detail. Quest also lets you specify **actions** for an object. Whereas a property lets you associate some text, an action lets you associate a script with an object.

This can be very powerful, and is useful if you’re creating large and complex games. You could, for example, set up actions relating to what happens when a magic spell is cast on that object. Then, when the player casts a spell, you can run that action using “Run an action script” from the “Run script” category.

To set up an action for an object, select it and click the Advanced tab. Click the “Edit Properties and Actions” button and then click the Actions tab. Here you can add as many actions as you like and enter a script.

22.6 Inherited Types

Inherited types give you a way to apply a group of properties and actions to several objects.

To set up a type, select Object Types from the tree menu. Here you can add, edit and remove the types that are set up in the game. When you're done, you can apply the type you've created to an object from the "Edit Properties and Actions" window.

23 What to Do When Things Go Wrong

As you develop your games, there will be times when things happen which you didn't expect – usually because you've forgotten to set something up, or you've made a mistake in one of your script commands.

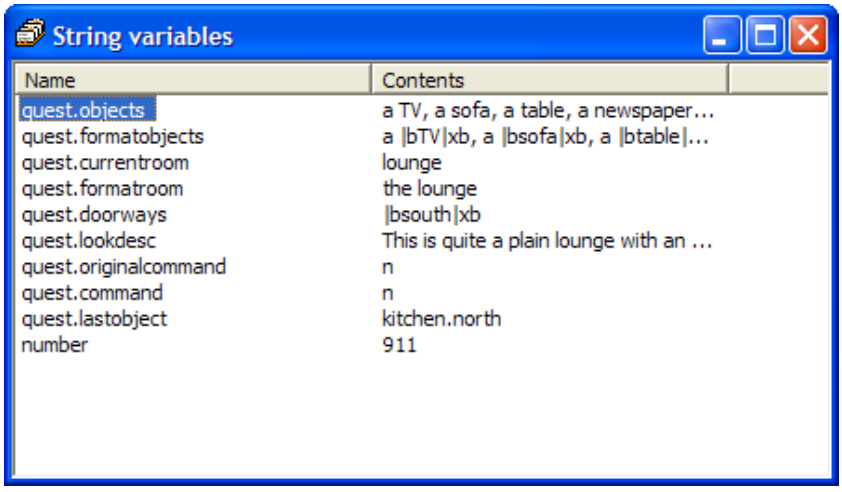
Fortunately, Quest provides you with a number of ways to keep an eye on what's going on inside your game while you're testing it. All of these are available from the Debug menu.

23.1 ASL Log

One of the first places to look if something strange happens is the ASL Log. Click the Debug menu and select ASL Log Window to take a look at the log for the current game. If you've made a mistake in your script commands (for example, if you've referred to an object which doesn't exist), Quest should tell you about it here.

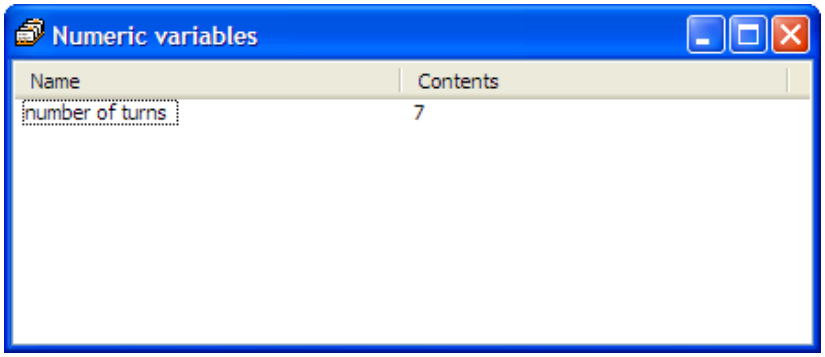
23.2 String Variables and Numeric Variables

While running our example game in Quest, click the Debug menu and select String Variables. You will see a screen like this:



This will show you the contents of all the string variables which have been set up. This includes both Quest's internal string variables, and variables that you've set up.

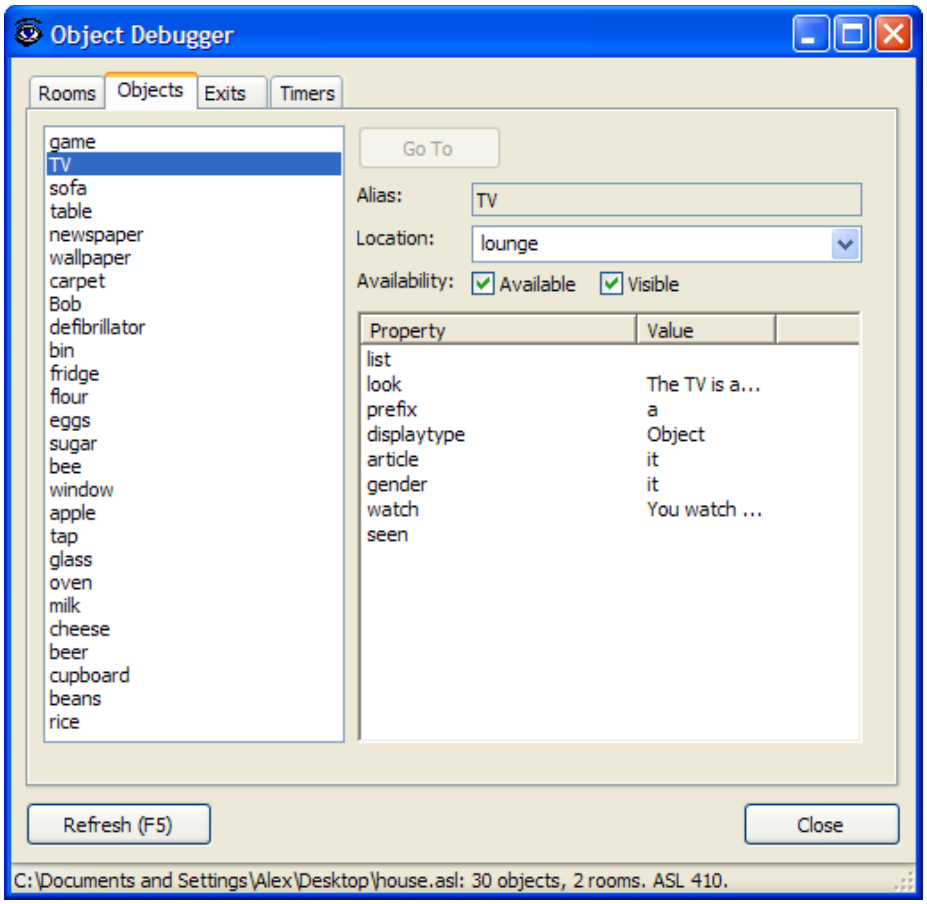
If you select Numeric Variables from the Debug window, you will see a similar view of all the numeric variables that are set up in your game:



This can be useful in diagnosing problems if your script commands aren't quite working the way you expect them to.

23.3 Object Debugger

The Object Debugger shows you all the objects and rooms in your game, together with all of their properties and whether they are hidden or unavailable. It also gives you information on all of the timers that you've set up.



The Object Debugger gives you a lot of power over your game, as you can make temporary changes to it while it is running:

- To jump between rooms, select that room and click "Go To"
- To move an object, select it and choose a new location from the drop-down list
- To change properties of an object or room, double-click that property and enter the new property information. You can add a property just by typing it in, and you can unset a property by typing "not" followed by the property name.
- You can view or change flags in the same way as properties by selecting "game" at the top of the object list.
- You can turn timers on or off by selecting them and toggling the "Active" checkbox.
- You can change a timer's interval (how often it fires) by selecting it and typing in a new value.

This can be useful if you have a large game and need to change something temporarily so that you can test it – it can be easier than restarting or reloading the game to test something.

Note that any changes you make in the Object Debugger are only temporary – you will not be changing your game's ASL file when you make updates here.

Of course, the huge power of the Object Debugger gives a great potential for your players to cheat. To stop this, select the “Disable debug windows” from the Game setup screen before releasing your game.

23.4 Using Transcripts

When playing a game, you can use **transcript** commands to record what you type to a file. You can then play back these commands at any time. This is useful for example for checking that when you make an update to your game, you don't make it unwinnable.

To turn on transcripting, type `SCRIPT ON` when playing the game. Any commands you type will then be placed into a text file, in the same folder as your game. The transcript file also records any selections you make from disambiguation menus or selection block menus.

If your game is called `mygame.asl` then the transcript will be called `mygame.asl.transcript`. You can view and edit the transcript file using a text editor such as Notepad.

To turn off transcripting, type `SCRIPT OFF`.

To run your transcript, type `RUN SCRIPT`. All the commands in your transcript file will then be run.

24 Quest Compiler

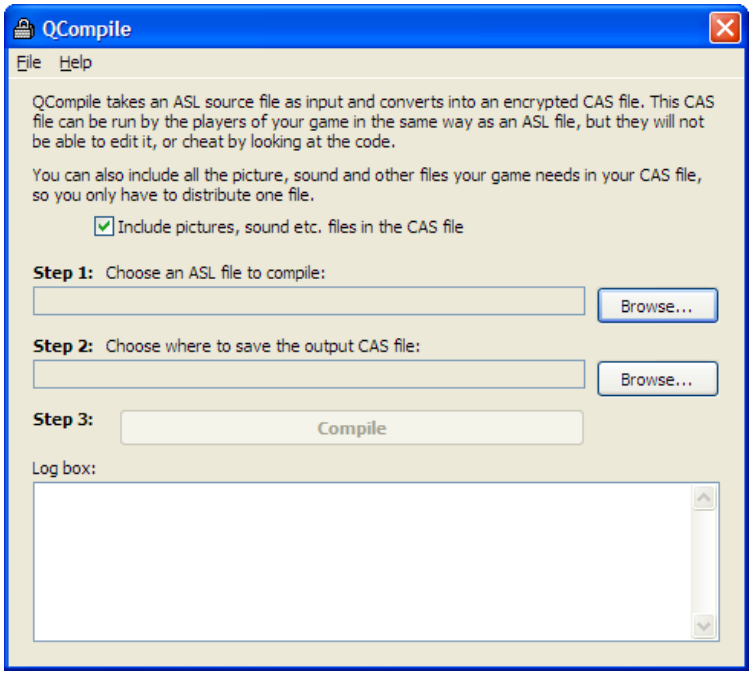
The Quest Compiler (QCompile) takes the ASL file you've created in QDK and turns it into an encrypted CAS file which you can give to other people to let them play your game. The CAS file can also include all the pictures, sounds and other files that your game needs.

A CAS file can't be edited in QDK, so you know that other people won't be able to edit your game or cheat (for example, if you've turned off debug windows, players won't be able to just load it up in QDK to turn them back on again).

24.1 Using QCompile

The easiest way to compile your game is directly from QDK. Click the File menu and then "Export CAS File", or click the toolbar button  . You will be prompted to choose a filename and where you want to save it, and then your CAS file will be created.

Alternatively, you can run QCompile by selecting Quest Compiler from your Start Menu.



This allows you to browse for an ASL file and choose where to save the CAS file. You can also specify whether or not to include pictures and sounds within the CAS file.

24.2 Including Additional Files

When you select the option to include pictures and sounds within your CAS file, QCompile will scan your ASL file looking for references to external files. These will then be automatically picked up and added to your CAS file.

Occasionally, you may need to tell QCompile about external files that need to be included. You only need to do this if there is some reason that it wouldn't pick up your external files automatically, for example, if you use string variables to tell Quest which picture or sound to display.

To manually add files, in QDK select Resources from the tree menu. You can then add a list of filenames, and QCompile will include these files when it generates your CAS file.

Because Quest extracts files from a CAS file only when required, if you've included a file manually you will also need to tell Quest when it needs to extract it. To do this, you need to use the "Extract a file from CAS" script command (in the Advanced category) before whichever script command you're using to show, display or call the external file.

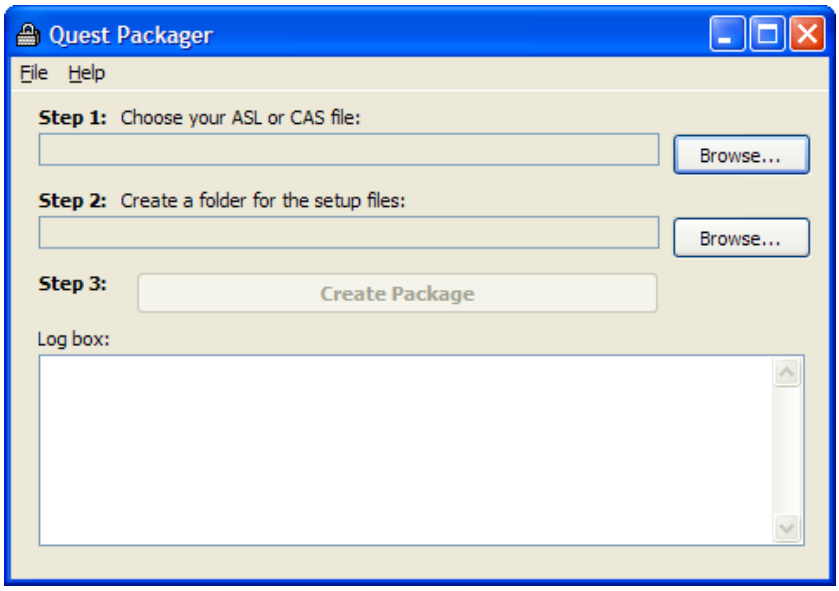
25 Quest Packager

Quest Packager builds on Quest Compiler, in that it not only encrypts your ASL file to stop people from editing it, but it also creates a setup EXE file for you, so you can distribute your game just like any other Windows application.

Your users won't have to download or install anything separately – they just install your game from its own EXE file, and that's it!

25.1 Using Quest Packager

Run Quest Packager from your Start menu.



It's very simple to use – just select your ASL or CAS file, choose a temporary folder for the setup files, and click the "Create Package" button.

After a minute or two, Windows Explorer will open the location of your new setup EXE file.

At this point you might want to rename it from "setup.exe" to something more meaningful (like "my game setup.exe"). You're now ready to send the file to your users, burn it to a CD, upload it to a website, etc.

The setup EXE will install your game on any PC running Windows 98 or later – no other downloads are required.

26 Releasing your Game

When you have finished creating a game of your own, you will want people to play it. In this section we will look at what you need to check before you release your game, and some advice on getting the game "out there".

26.1 Before Releasing

Before you even think about releasing your game, you need to thoroughly check to make sure it works properly, and that it has sensible responses for things that a player might reasonably type while playing it. To create a good game is a lot of hard work, and while it might be tempting to release your first efforts after a minimal amount of testing, your players won't thank you for it.

And no matter how tempting it might be, do not even *think* about releasing the game you've created while working your way through this tutorial!

Here are some things to think about before unleashing your game on an unsuspecting public:

- Think about all the objects a player might refer to – make sure everything you refer to in your descriptions is at least set up as a scenery object.
- Think about all the different things a player might reasonably try to do with an object – set up verbs, even if they just tell the player that they can't do that.
- Think about all the different ways a player might type a command, and make sure you have enough verb alternatives set up.
- Think about the different ways a player might refer to the same thing, and set up Other Names.
- Make sure you test your game thoroughly. Get some other people to test it – you'll be surprised at all the things they pick up that you would never have thought of.

26.2 Creating the Game Package

When your game is finally finished, make sure you turn off the debug windows from the Game setup screen – you don't want players to be able to cheat that easily, do you?

The next step is to convert your game into a CAS file. If you choose the option to include pictures and sounds within the file, the only file you then need to distribute is the CAS file it creates.

If you want players to be able to play your game without them having to download Quest as well, turn your game into a setup EXE file using Quest Packager.

26.3 Uploading the Game

There are various places you can upload your game, so that other people can get hold of it:

- textadventures.co.uk: http://www.textadventures.co.uk/. Hundreds of people visit this website every week looking for adventure games to play, and they can play the games online without having to download any software.
- The IF Archive at www.ifarchive.org.
- Or of course, you can upload it to your own webspace if you have any.

And if you've packaged your game using Quest Packager, the world is your oyster! You can submit your game to one of the many websites that accept Windows applications and games. One of the biggest is download.com, which you can submit to via its sister site upload.com.

26.4 Announcing your Game

Once you have uploaded your game, you will want people to know where to get hold of it, and how to play it.

Your users will need to download Quest from http://www.axeuk.com/quest to be able to play the game. The Quest player is a free download.

You can announce your game on the Quest Games forum at http://www.axeuk.com/phpBB3/viewforum.php?f=5.

You can also announce it on the rec.games.int-fiction newsgroup: http://groups.google.com/group/rec.games.int-fiction.

27 Answers to Exercise

27.1 Defibrillator

There are various ways you could do this, but one of the simplest things to do is to update the defibrillator's "use" script so it reads:

```
If "Bob" has the property "dead"  Then {
Modify "Bob"'s property: "not dead"
Print "Miraculously, the defibrillator lived up to its promise, and Bob is now alive again. He says his head feels kind of fuzzy." } Else Print "You've already revived Bob - he probably wouldn't like it much if you used the defibrillator on him again."
```

27.2 Tap Exercise

Here's one way of completing the exercise in chapter 15.

In this method, I've given the tap object a property of "running", which is set to on after the player turns on the tap. I've given the glass a property of "full", which is set after the player fills the glass.

27.2.1 Tap Object

To the tap object I added the verb "switch on", which has "turn on" as an alternative form. This runs a script:

```
If "tap" does not have the property "running"  Then {
Print "You turn on the tap."
Modify "tap"'s property: "running" } Else Print "The tap is already on."
```

To add "turn off", I used the "switch off" verb, which has "turn off" as an alternative. I made "switch off" run this script:

```
If "tap" has the property "running"  Then {
Print "You turn off the tap."
Modify "tap"'s property: "not running" } Else Print "The tap is already turned off."
```

The tap's "look" description is also a script:

```
If "tap" has the property "running"  Then Print "The tap is running." Else Print "Just an ordinary tap."
```

27.2.2 Glass Object

To the glass object I added a verb "fill", which runs this script:

```
If "tap" has the property "running"  Then {
Print "You fill the glass with water."
Modify "glass"'s property: "full" } Else Print "You need to turn the tap on first."
```

I also added a "drink" verb, which runs this script:

```
If "glass" has the property "full"  Then {
Print "You drink the water. How refreshing!"
Modify "glass"'s property: "not full" } Else Print "You can't drink from an empty glass!"
```

The glass's "look" description is this script:

```
If "glass" has the property "full"  Then Print "This glass is full of water." Else Print "This glass is empty."
```

I also gave the glass object an alternative name of "water", so the player can type "drink water" as well as "drink glass", which seems a bit more natural.

27.2.3 Testing the Script

Here's some example game output using the above script:

```
> look at glass
```

This glass is empty.

> look at tap

Just an ordinary tap.

> turn on tap

You turn on the tap.

> fill glass

You fill the glass with water.

> look at tap

The tap is running.

> turn off tap

You turn off the tap.

> look at glass

This glass is full of water.

> drink water

You drink the water. How refreshing!

> look at glass

This glass is empty.

> fill glass

You need to turn the tap on first.

This works nicely – and the player can keep on turning the tap on and off, filling the glass and drinking the water, over and over again. Unfortunately we've not implemented a toilet in our game…

28 Index