

JACL

Adventure Creation Language

Author's Guide

Sixth Edition

JACL Interpreter v2.7
<http://code.google.com/p/jacl>

JACL Interpreter copyright 1992-2010 Stuart Allen
Please send any bugs or feature requests to stuartallen1972@gmail.com

Webserver and preprocessor code copyright 2001-2002 Andreas Matthias.

Internationalisation by Niels Haedecke, Thomas Schwarz and Eric Forgeot.

Javascript, CSS and user registration code by Thomas Schwarz.

Glk libraries by Andrew Plotkin, David Kinder, Tor Andersson and Ben Cressey.

Special thanks to Robert Osztolykan, Parham Doustadar, Jose Lacal and Eric Forgeot for their testing and editing.

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

Table of Contents

Introduction.....	1
Typographical Conventions.....	3
Installation.....	5
Compilation.....	5
Playing the Sample Games.....	6
cgijacl and fcgijacl.....	7
Playing the Sample Games.....	9
Using JACL with FastCGI and Apache.....	10
Playing Interactive Fiction.....	13
Moving Around.....	13
Manipulating Objects.....	13
Interacting with Characters.....	15
Meta Commands.....	16
Tutorial Game.....	17
Language Syntax.....	17
Program Structure.....	17
Getting Started.....	18
Locations.....	18
The Player.....	21
Some Introductory Text.....	23
Objects.....	24
Verbs and Functions.....	26
Overriding Functions.....	27
Doors.....	28
Non-player Characters.....	32
The Passing of Time.....	34
Winning and Losing the Game.....	35
Testing, Debugging and Releasing.....	39
The WALKTHRU Command.....	39
Transcripts.....	40
The Debug Library.....	41
The INSPECT Command.....	41
The Verb VALUEOF.....	41
The Verb FETCH.....	42
Releasing Your Game.....	43
Screen Display.....	45
The WRITE Command.....	45
Special Characters.....	46
Printing the Value of Variables.....	46
Printing the Value of Item Elements.....	46

Table of Contents

Screen Display

Printing the Names and Descriptions of Objects.....	47
Sentences Referring to Varying Objects.....	47
Custom Macros.....	48
Printing the Value of Strings.....	49
The PRINT Command.....	49
The LOOK Command.....	50
The MORE Command.....	51

Glk and Multimedia.....53

Blorb Files and the bjob Utility.....	53
The IMAGE Command.....	55
The SOUND Command.....	55
The VOLUME Command.....	56
The STOP Command.....	56
The TIMER Command.....	56
The STYLE Command.....	57
The Status Window.....	57
The UPDATESTATUS Command.....	60

HTTP and HTML.....61

Document Structure.....	61
Linebreaks.....	63
The Player's User ID.....	63
The Player's Commands.....	64
Ajax Requests.....	64
The BUTTON Command.....	65
The HYPERLINK Command.....	65
The CONTROL Command.....	65
The OPTION Command.....	66
The IMAGE Command.....	67
The Media File.....	67

GUI Web Interface.....69

Using the Interface.....	70
--------------------------	----

Flow Control.....71

The IF, IFALL and ENDIF Commands.....	71
The IFSTRING Command.....	72
The IFEXECUTE Command.....	73
The ELSE Command.....	73
The LOOP and ENDLOOP Commands.....	74
The SELECT and ENDSELECT Commands.....	75
The REPEAT and UNTIL Commands.....	76
The WHILE and ENDWHILE Commands.....	77
The RETURN Command.....	77

Table of Contents

Changing Data.....	79
The SET Command.....	79
Type Casting.....	80
The SETSTRING and ADDSTRING Commands.....	81
The PADSTRING Command.....	81
Movement.....	83
The MOVE Command.....	83
The TRAVEL Command.....	83
Moving Non-player Characters.....	85
The DIR_TO and NPC_TO Commands.....	85
Special-Purpose Commands.....	87
The POINTS Command.....	87
The PROXY Command.....	87
Trigonometry.....	88
The POSITION Command.....	88
The BEARING Command.....	88
The DISTANCE Command.....	89
The ASKNUMBER AND GETNUMBER Commands.....	89
The GETSTRING Command.....	89
The GETYESORNO Command.....	90
The SAVEGAME and RESTOREGAME Commands.....	90
The TERMINATE Command.....	91
The UNDOMOVE Command.....	91
Attributes.....	93
The ENSURE Command.....	93
Object Attributes.....	93
Location Attributes.....	94
User Attributes.....	95
Functions.....	97
The EXECUTE and CALL Commands.....	97
Passing Arguments to a Function.....	99
The function-call count.....	100
The RETURN Command.....	101
Responding to the Player's Moves.....	101
Special Functions.....	104
Utility Functions.....	106
Creating New Verbs.....	107
Pointers.....	111
Object Pointers.....	111
Location Pointers.....	111

Table of Contents

Object Resolution.....	113
Object Naming.....	113
Disambiguation.....	114
Definitions in Detail.....	117
Objects.....	117
Locations.....	120
Integer Variables.....	122
Internal Integer Variables.....	123
String Variables.....	124
Arrays.....	124
Constants.....	125
Synonyms.....	125
Filters.....	126
Grammar Statements.....	126
User Attributes.....	128
Parameters.....	128
CSV Files.....	131
The ITERATE and ENDITERATE Commands.....	131
The UPDATE, ENDUPDATE and INSERT Commands.....	132
The APPEND Command.....	133
Internals.....	135
Constants and Random.....	135
Internal Commands.....	135
The Menu Library.....	137
CSV Files.....	139
The ITERATE and ENDITERATE Commands.....	139
The UPDATE, ENDUPDATE and INSERT Commands.....	140
The APPEND Command.....	141
Appendix A: JACL Attributes.....	143
Appendix B: Library Verb Functions.....	145
Appendix C: Tutorial Game Source Code.....	149
Glossary.....	155

Introduction

JACL is a language for writing interactive fiction, also known as text adventure games. It is easy to learn and comes with an extensive library of game verbs that will allow you to create the beginnings of your first game quickly and easily. The [tutorial chapter](#) of this guide walks you through the creation of a simple, but complete text adventure game and demonstrates all the main principles of the JACL language.

JACL is an interpreted language, so once you have written your games they can be run without modification on any platform that has a JACL interpreter. The JACL interpreter uses the Glk API to talk to its user interface which allows it to be compiled with many different Glk implementations on many different operating systems. Although interactive fiction is primarily a text-based medium, many of these Glk interfaces allow you to also fill your games with pictures and sound.

JACL also has two web-enabled interpreters that are easy to configure and run. If desired, games that do not require a complex graphical interface can be written in such a way that the same game file can be run by both the console-based and web-based interpreters. If desired, however, the web-enabled interpreters can be used to create a variety of turn-based games with interfaces vastly different to interactive fiction. The Blackjack game included with this distribution is one such example.

The latest version of JACL can be found at <http://code.google.com/p/jacl/>.

I would also like to take this opportunity to thank some of the people who have contributed to this system: Robert Osztolykan and Parham Doustdar for their invaluable beta testing, editing and suggestions; Andreas Matthias for contributing the internal web server and preprocessor code; Thomas Schwarz for his Javascript, CSS and user registration code; David Fisher for contributing the route-finding code; Niels Haedecke, Thomas Schwarz and Eric Forgeot for their work on translation and internationalisation; Andrew Plotkin, David Kinder, Tor Andersson and Ben Cressey for their Glk libraries. I would also like to thank Ben Cressey for his assistance in adding Unicode support to JACL.

Stuart Allen
Sydney, Australia
April 2010
stuartallen1972@gmail.com

Typographical Conventions

The use of **bold** indicates that this word is used verbatim in the appropriate manner.

The use of *italics* indicates text that should be substituted with the appropriate value.

The use of `typewriter` font indicates code segments, typed commands or transcripts.

Underlined text is occasionally used for emphasis.

Text enclosed in square brackets, [such as this], indicates an optional parameter.



The warning graphic indicates a potential area for error. Special care should be taken when working in these areas.



The information graphic indicates advanced, non-essential information that may confuse the novice author. These may be safely skipped until you have a greater level of experience.

Installation

The latest version of JACL can be found at <http://code.google.com/p/jacl>. JACL is available as a zip file for Microsoft Windows and tar file for Unix operating systems. Once you have extracted the appropriate archive you will have a directory called **jacl** with several subdirectories. You will find the executable files in the **bin** subdirectory. In the Linux distribution you will find four interpreters: **jacl** is a console interpreter that uses the GlkTerm library by Andrew Plotkin, **garjacl** is a graphical interpreter that uses the Gargoyle library by Tor Andersson and Ben Cressey, while **cgijacl** and **fcgijacl** are the two web-based interpreters. **cgijacl** can be run directly from the command line and is used more for development purposes while **fcgijacl** must be used with a FastCGI-enabled webserver and is used more for production deployments. You will also find the program **bjorb**, a small utility (based on **blc** by Ross Raszewski) for creating the Blorb resource files that will contain any images and sounds you wish to use in your games. In the Microsoft Windows distribution you will find a single interpreter called **garjacl** and the same **bjorb** utility as in the Linux version.

Compilation

The **src** directory contains the C source code to the JACL interpreter. Also included is the source to **glkterm** a Glk implementation by Andrew Plotkin based on ncursesw, and the **bjorb** utility .

Interpreter	Required Libraries
jacl (Posix)	GlkTerm by Andrew Plotkin (included) and ncursesw.
garjacl	Gargoyle by Tor Andersson (included as a DLL and .so shared library).
cgijacl	No special libraries are required to compile cgijacl , however a POSIX compliant system is needed. FastCGI, available from http://www.fastcgi.com/index.html , or as the Debian packages:
fcgijacl	<code>libfcgi-dev</code> <code>libfcgi0c2</code>

To compile the source for the JACL interpreters and the **bjorb** under Linux simply change to the **src** subdirectory and type the following command:

```
make install
```

This will compile all the relevant programs and copy the executable binaries into the **bin** subdirectory.

All the extra DLLs required for Microsoft Windows are included in the **bin** directory. In Linux distributions, only the core Gargoyle share library is included. This is file **libgarglk.so** in the **src/Gargoyle** directory and should be copied to the directory **/usr/local/lib**. Other required libraries should be installed using your operating systems packaging system. Below is the output of **ldd** to showing the complete list of libraries required:

libgarglk.so	libjpeg.so.62
linux-gate.so.1	libm.so.6
libc.so.6	ld-linux.so.2
libfreetype.so.6	libz.so.1
libgtk-1.2.so.0	libgdk-1.2.so.0
libXi.so.6	libXext.so.6
libX11.so.6	libglib-1.2.so.0

The JACL Author's Guide

libpng12.so.0	libSDL_mixer-1.2.so.0
libSDL-1.2.so.0	libgmodule-1.2.so.0
libdl.so.2	libXau.so.6
libxcb-xlib.so.0	libxcb.so.1
libasound.so.2	libdirectfb-1.0.so.0
libfusion-1.0.so.0	libdirect-1.0.so.0
libpthread.so.0	libXdmcp.so.6

In the **games** directory you will find several JACL games. A JACL game consists of one or more **.jacl** files and an optional **.blorb** file that contains multimedia resources used by the game. If a JACL game consists of more than one **.jacl** file, the other file will be one of the library files from the **include** directory.

Inside the **games** directory you will also find a **temp** directory. This is the directory where the interpreter will create the **.j2** files. A **.j2** file is a concatenation of a main game file and all other files included from that file. They are created by the interpreter when the game is first run and are encrypted by default.



Under Unix, the user the JACL interpreter is run as must have write permissions to the **temp** directory.

The **include** directory is found inside the **games** directory. In the **include** directory you will find **verbs.library**, the main interactive fiction library and **frame.jacl**, a skeleton program used as a starting point when writing a new piece of interactive fiction.

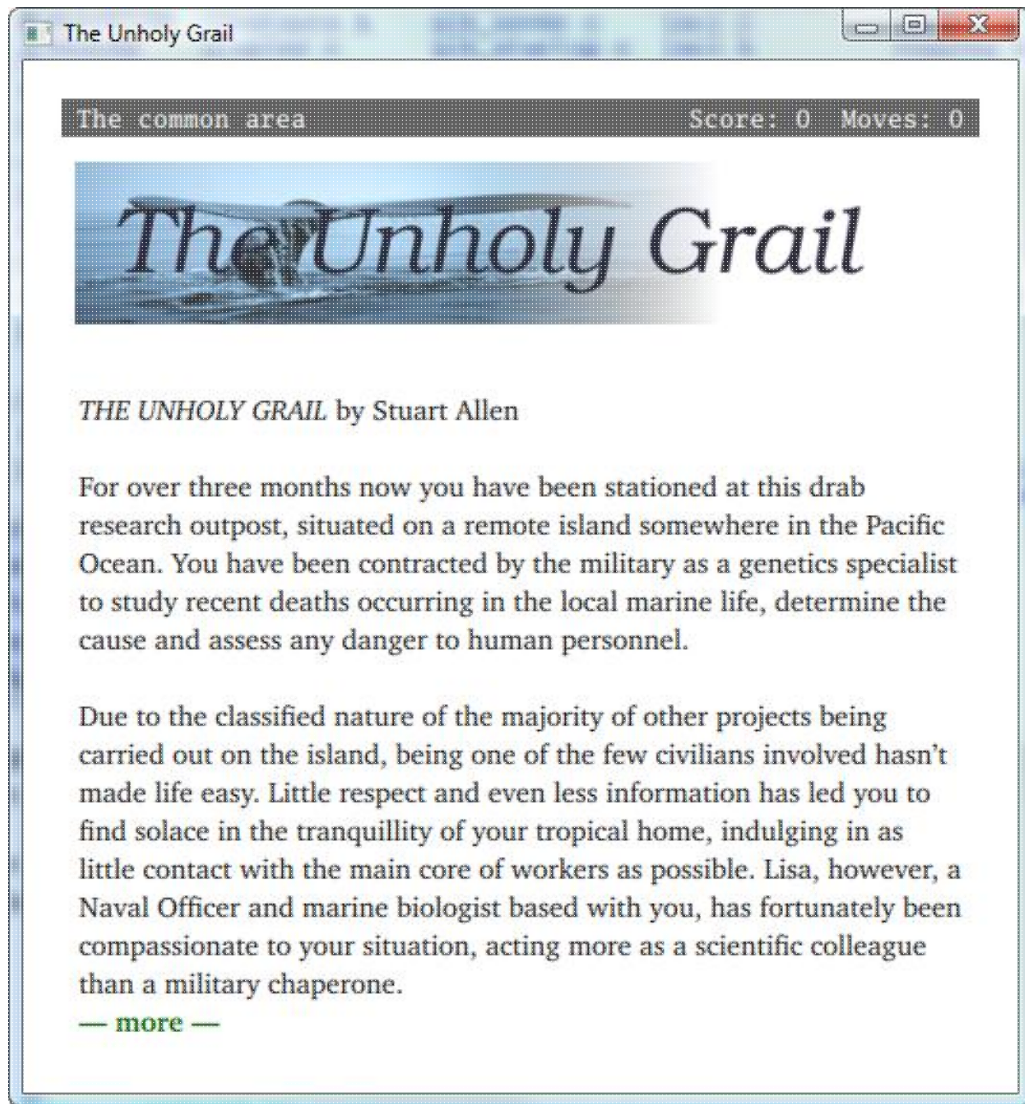
In the **guide** directory you will find this documentation.

Playing the Sample Games

Once you have compiled versions of the JACL interpreters, try running one of the provided games by changing to the **bin** directory and typing the command:

```
./garjacl ../games/grail.jacl
```

When you do so, you should see something like the screen below:



cgijacl and fcgijacl

When the **cgijacl** and **fcgijacl** interpreters start they will first look for their configuration file in the current directory. If this does not exist they will look for it in the **etc** directory that is at the same level as the **games** directory. In other words, from the directory the game file being run is in, the **jacl** interpreter will look for **../etc/cgijacl.conf**. If neither of these files is found, the **cgijacl** or **fcgijacl** interpreters will also look for the file **/etc/cgijacl.conf**.

The configuration files for **cgijacl** and **fcgijacl** may contain any of the following lines:

Directive	Description	Default
access_log <i>FileName</i>	This indicates the file that should be used to log all moves made while playing. If a directory is specified with the trailing forward slash, the file name access.log is appended to the supplied directory.	../log/access.log (from the location of the game file)
error_log <i>FileName</i>		

The JACL Author's Guide

This indicates the file that should be used to log all errors that occur while JACL programs are running. If a directory is specified with the trailing forward slash, the file name **error.log** is appended to the supplied directory.

../log/error.log (from the location of the game file)

include *Directory*

This indicates the directory to look in to find any file specified in a **#include** preprocessor directive.

A directory called **include** beneath the directory the game file is stored in. If an included file is not found in this directory, the directory the game file is stored in is searched.

temp *Directory*

This indicates the directory to store the processed version of the game file.

A directory called **temp** beneath the directory the game file is stored in. If this directory does not exist, the **.j2** file will be created in the same directory as the game file.

Below is an example **cgijacl** configuration file:

```
# "cgijacl.conf"
# CGIJACL interpreter configuration file.

access_log      "/usr/local/jacl/log/error.log"
access_log      "/usr/local/jacl/log/access.log"
temp            "/usr/local/jacl/games/temp/"
include         "/usr/local/jacl/games/include/"
```

Playing the Sample Games

Once you have compiled the JACL interpreters, try running one of the provided games by changing to the **bin** directory and typing the command:

```
./cgijacl ../games/blackjacl.jacl
```

This command will start the JACL interpreter on the default port of 4269, resulting in the following output:

```
JACL Interpreter v2.7.0
WebJACL: Registered 8 media.
WebJACL: No port number specified (-p ), using default port 4269.
WebJACL server configured on WOPR:4269
```

To play this game, open your favourite web browser and navigate to the URL **http://localhost:4269**. When you do so, you should see something like the screen below. If not, see the section on [Trouble Shooting](#).



To play this same game from the console, try the following command from the **bin** directory:

```
./jacl ../games/blackjack.jacl
```

Using JACL with FastCGI and Apache

While the **cgijacl** interpreter is able to respond to HTTP requests internally, the **fcgijacl** interpreter is designed to be used with a FastCGI-enabled webserver such as Apache. The **cgijacl** interpreter is very easy to use and is ideal for developing a new game or allowing a small number of users to play your completed games. The **fcgijacl** interpreter, when used with a production-quality web server such as Apache, is probably better choice for situations where a high level of traffic is expected.

Before the **fcgijacl** interpreter can be successfully used with an Apache server, the **mod_fastcgi** module must be installed and configured. There are many different ways to install both the Apache server and the **mod_fastcgi** module, all of which are detailed in their respective documentation. **mod_fastcgi** is available from http://freshmeat.net/projects/mod_fastcgi or as the Debian package:

```
libapache2-mod-fastcgi
```


The JACL Author's Guide

FastCGI is a system for interfacing a web server with an external binary program, in this case the JACL interpreter. With normal CGI, a program is started to handle each individual request. The program executes, produces some output then terminates. Depending on the startup time of the specific program this can be an extremely inefficient process. With FastCGI, programs are run once then wait in a loop, processing requests as they are made. This suits programs such as the JACL interpreter perfectly as it has a fair amount of processing to do when a game is first run, yet very little for an individual move made by a player.

Once **mod_fastcgi** is installed, Apache requires a file called **fastcgi.load** in your **mods-enabled** directory that contains the line:

```
LoadModule fastcgi_module /usr/lib/apache2/modules/mod_fastcgi.so
```

To configure FastCGI, you must edit the file **fastcgi.conf** in your **mods-enabled** directory to read:

```
<IfModule mod_fastcgi.c>
  AddHandler fastcgi-script .fcgi

  ScriptAlias /fastcgi-bin/ "/usr/local/fastcgi-bin/"

  <Location /fastcgi-bin>
    SetHandler fastcgi-script
    Options ExecCGI
  </Location>

  FastCgiIpcDir /var/lib/apache2/fastcgi
</IfModule>
```

This configuration creates a script alias of **/fastcgi-bin** that points to the directory **/usr/local/fastcgi-bin**. This directory must be created so that the user that the Apache server runs as has read and execute permissions to it. A script alias is a way of mapping URLs to directories on your file system. The above script alias says that any resource being accessed using a URL starting with **/fastcgi-bin** should be looked for in the directory **/usr/local/fastcgi-bin**.

The location block says that any scripts accessed within the **/usr/local/apache/fastcgi-bin** directory are to be handled by **mod_fastcgi** and that the execution of CGI scripts is permitted. As you have probably guessed, this is directory where all your JACL games will be placed. You may alternatively wish to set a script alias like **/fastcgi-jacl** to point to the directory **/usr/local/jacl/games/** or wherever your JACL games reside.

The **images** directory beneath the **games** directory of your JACL distribution will need to be copied to the Apache document root. The sample games are designed to read the images from **/images** when played with **fcgijacl**. It is necessary to load images from somewhere beneath the document root as Apache attempts to run any resource accessed from beneath **/fastcgi-bin** as a FastCGI script, which the images are not.

Next, the file **cgijacl.conf** must be moved from **jacl/etc** to **/etc**. This file contains several configuration details, the most important directive for **fcgijacl** is **temp**. This specifies the directory that all persistent data will be written to. By default this is specified in **cgijacl.conf** as **/usr/local/jacl/games/temp**. It is important to ensure that the user the Apache server runs as has read and write permissions to whichever directory you specify by using a command such as:

```
chmod 777 /usr/local/jacl/games/temp
```

Regardless of where you locate your JACL games, you will need to edit the first line of each game to point to the **fcgijacl** interpreter. For example:

The JACL Author's Guide

```
#!/usr/local/jacl/bin/fcgijacl
```

The below line will appear in the Apache error log if the first line of the game points to the interpreter in the wrong location. The line can be misleading as it is not the file **grail.jacl** that cannot be found but the interpreter binary it is attempting to run itself using.

```
FastCGI: can't start server "/usr/local/fastcgi-bin/grail.jacl" (pid 2539), exec  
le() failed: No such file or directory
```

Once you have finished configuring JACL and Apache, start Apache and open the following URL from your browser: **<http://localhost/fastcgi-bin/tutorial.jacl>**. If all has gone well, you should see the title page for this game. If not, try looking in Apache's **error_log** and **/usr/local/jacl/log/error.log**.

Playing Interactive Fiction

Interactive fiction involves the reader in a way that static stories do not. Rather than reading sequentially from beginning to end, you control the actions of the main character by typing commands in the form of simple English sentences. Before setting out on your adventure there are a few concepts that must be explained in order for you to interact successfully with the virtual world created by the author.

Moving Around

The physical space that you will be exploring is divided up into discrete units called locations. Locations may be travelled between using the following commands:

Command	Direction
n	North
s	South
e	East
w	West
ne	Northeast
nw	Northwest
se	Southeast
sw	Southwest
u	Up
d	Down
in	In
out	Out

The description of each location will tell you the directions in which you can travel from it. Be aware, however, that some exits may only be available under certain circumstances such as while a door is open or a bridge lowered.

Manipulating Objects

Within these locations you will find many objects that you can interact with. These objects may be referred to by as few or as many of their names as is required to uniquely identify it. For example, if the story informs you that "There is a silver key resting here." Then it may be referred to as either **silver**, **key** or **silver key**. If your reference is ambiguous, such as using **key** when there is both a silver and a gold key in the current location then you will be prompted to be more specific.

The next thing to learn about interactive fiction is the way in which you interact with these objects.

The JACL Author's Guide

The basic sentence structures understood by the game are:

Command Structure	Example command
<i>action</i>	scream
<i>action object</i>	take book
<i>action object preposition object</i>	unlock door with key

An action does not necessarily have to be a single word. For instance, "examine rock through magnifying glass", an example of the last syntax, may also be expressed as "look at rock through magnifying glass". Following is a list of some of the other actions that can be used to interact with the objects you find:

turn	read	pull	push	eat	cut
lock	unlock	rub	attack	taste	drink
break	smell	listen	pour	move	light
play	blow	throw	wear	open	close
take	drop	insert	remove		

Where an object is expected, the words **it**, **them**, **him** and **her** can be used to indicate the last appropriate object referred to in one of your commands. With some commands it is possible to supply more than one object. Multiple objects can be specified in a multitude of ways. The words **all** or **everything** will tell the interpreter that you wish to perform the action on all the objects in the appropriate scope. For example, with the command **drop all**, the interpreter will perform the **drop** action for every object you are currently carrying.

Any object reference can be qualified by using the word **from**. Although this can be used at any time, it is most useful in conjunction with **all** to construct commands such as **take all from chest**. This command will perform the **take** action for every object that is currently inside the chest.

The word **except** can be used once per list of objects, with any object to the right of the word **except** being removed from list of objects to the left. An example of this is the command **take all from chest except the gem**. The important thing to be aware of with the word **except** is that any objects supplied after the word **except** are objects the action is not to be performed on. With this in mind you can see how the two commands below have very different results.

```
>take sword and all from chest except gem
>take all from chest except gem and sword
```

With the first command the sword will be taken, with the second command it will not. So in summary, any list of objects may use the word **from** many times, but **except** only once at most. Individual objects can be separated by the word **and** or a comma (,) while the word **then** or a period (.) is used to separate completely different commands. Below is a legal command that uses all of the forementioned functionality:

```
>take sword, shield, all from chest and all from shelf except gem then go north
```

When playing interactive fiction, it is important to have a general feeling for how much can be achieved with a single command. A task such as defusing a bomb would generally not be performed by typing **defuse bomb**. The following is a sample transcript to give you a better feel for how a task such as this might be

The JACL Author's Guide

achieved:

>examine bomb

The bomb is about a foot square and has a small panel in its upper surface. The panel is currently closed.

>listen to bomb

The bomb is making an ominous ticking sound.

>open panel

You open the panel to reveal a red wire, a green wire, a blue wire, a clock and some dynamite.

>i

You are carrying a bomb manual, an insurance policy and some wire cutters.

>read manual

Leafing through the book you come to the page on diffusing. Seems straight forward enough, just cut the blue wire then the red one.

>cut blue wire with cutters

Reaching carefully in to the bomb's casing you cut the blue wire. So far so good.

>cut red wire with cutters

Holding your breath you snip the red wire.

>listen to bomb

The bomb, thankfully, is now silent.

As you can see, each task you wish to achieve must be broken down into its component actions.

Interacting with Characters

During the course of the story you may encounter other computer controlled characters. These characters may be interacted with using the following commands:

talk to *character*

give *object to character*

show *object to character*

ask *character for object*

ask *character about object*

tell *character about object*

Meta Commands

There are also some special commands for interacting with the JACL interpreter rather than the game world it is currently running:

Command	Action
verbose	Always describe locations when visited.
brief	Only describe locations when being visited for the first time.
restart	Restart the game from the beginning.
script	Start recording a complete transcript of your game session.
unscript	Stop recording a started transcript.
score	Display your current score.
look	Display the description for the current location.
save	Save the current game state.
restore	Restore a saved game state.
i	Display a horizontal list of possessions.
inv	Display a vertical list of possessions.
oops <i>NewWord</i>	Replace the erroneous word in the last command with the supplied new word.
undo	Take back the last command issued.
again	Repeat the last command issued.
info	Display interpreter version information.
about	Display game version information.

You are now ready to embark on your first adventure into the world of interactive fiction. Remember to make a map as you go and read everything carefully as vital clues may be hidden in the descriptions of the locations and objects you come across. Examining every object you can refer to is a good idea, as is saving your position often. Well, good luck, and above all have fun!

Tutorial Game

JACL is a language for writing interactive fiction. By providing short definitions for objects and locations you will rapidly build your virtual world. By associating small amounts of simple code with these objects and locations you will bring your world to life. During the course of this tutorial you will see the construction of a small text-only game from beginning to end. The full [source code](#) for this tutorial game is included in Appendix D and in the **games** subdirectory of your JACL distribution.

Language Syntax

JACL is an interpreted language. This means that the interpreter directly reads your program source code and executes it without the need for it to be compiled first. Each JACL command or definition must be on a line of its own and no single line may not exceed 1024 characters in length. Any single parameter that includes spaces, commas, colons or tabs must be contained within double quotes, as these are all considered to be white space. If a command accepts parameters, each parameter must be separated by white space.

Any text that appears between a hash (#) or a semicolon (;) and the end of the line is considered a comment and has no affect on the program's execution. Comments are useful to explain the purpose of your code to other authors or to refresh your own memory when returning to old code.

One special use of comments is to tell the command shell the location of the interpreter that can execute your program. The first line of most JACL programs will be something like:

```
#!../bin/jacl
```

If a JACL program has a first line similar to this, when it is executed directly the shell will pass the name of the program as an argument to the interpreter specified after the **#!**. This means that the following shell command:

```
./grail.jacl
```

becomes the equivalent to the following command:

```
../bin/jacl grail.jacl
```

This line presumes that all games are run from the **games** directory of your JACL distribution and that you are currently in that directory. To remove this restriction change the first line of your game to indicate the full absolute path to your JACL interpreter of choice. For example:

```
#!/usr/local/bin/garjacl
```



In order for this to work on Microsoft Windows, you must be running a Unix shell under Cygwin. To work under Unix, the JACL program must be set to executable using a command such as:

Warning `chmod 755 grail.jacl`

Program Structure

A JACL program consists of two fundamental components: data and code. Data is provided in the form of definitions, code in the form of functions. There are nine types of definition, their keywords being **location**,

object, synonym, filter, integer, constant, string, attribute and **grammar**. A function begins with a left curly brace (`{`) followed directly by its name, and ends with a right curly brace (`}`) on a line on its own. Any text that appears on the lines between the two curly braces is considered to be the code body of that function. Other than comments, the only text that can appear outside a function is one of the nine definitions. While definitions are read once at the start of the game then stored in memory, functions are executed from the game file as required during play.

Getting Started

To begin writing your first adventure game using JACL you will need to copy the file **frame.jacl** from the directory **jacl/games/include** to a file called **game.jacl** in the directory **jacl/games**. This new file is the one you will edit to create your game.

The file **frame.jacl** is a skeleton program that provides a good starting point when writing a piece of interactive fiction from scratch. The file **verbs.library**, is also included from the last line of **frame.jacl**. While the code in **frame.jacl** should be modified to suit your game, the code in **verbs.library** should never be modified. This allows new versions of **verbs.library** to be installed without adversely affecting your game. **verbs.library** contains a extensive set of verbs for use by the player during the game.

Now open **game.jacl**, the beginnings of our tutorial game, with your favourite text editor. The name **game.jacl** has been chosen for the purpose of this tutorial, your game file can be given any name you like.

At the top of this file you will find four constants. These constants are used to store bibliographical information about your game that can be directly read by cataloguing tools. For this reason the names of these constants, including the case, must be left unchanged which their values changed as appropriate. For information visit [The Treaty of Babel](#).

Locations

In text adventure games, a location represents a discrete physical space that the player and other objects can be in, such as a room in a house. Although objects can also be placed inside other objects, the player can only be in a location. We will now begin to define the first location of this tutorial game. To do this, insert the following line of code below the bibliographical constants:

```
location bedroom: master bedroom
```

This line states that we wish to define a **location** with the label **bedroom**. This label is the name by which we will refer to this location within our code. For those of you with previous programming experience, you can think of this label much in the same way as a variable name. Following the location's label is a space-delimited list of names by which the player can refer to this location during game play. All locations must have at least one name, and may have as many as can fit into a single line of code. In this example, the location has two names: **master** and **bedroom**. If you do not supply any names, the location's label will become its one and only name.

All the locations and objects you define have associated properties. When you define an object or location it is possible to set the initial value of these properties. For the new location **bedroom** we are going to set the initial value of two properties: **short** and **west**. This is done by adding two more lines to the location definition so it looks like this:

```
location bedroom: master bedroom
```


The JACL Author's Guide

```
short      a "master bedroom"
west       bathroom
```

short is a two-value property used to supply a short description of the location. This description is used by the library code when referring to the location in a sentence. The first parameter after the keyword **short** is its indefinite article, the second is its description. The supplied indefinite article is used to prefix the description when the **{list}** macro is used. This would normally be **a** or **an**.

The **{list}** macro is used as a parameter of a **write** command as a way of outputting text that refers to an object. It is normally used when referring to an object or location in a list of several objects or locations. The most common example of this is a list of objects that are inside another object or held by another object, such as a desk drawer or the player's inventory. Below is an example of the **{list}** macro:

```
write "The house you are in has " bedroom{list} ".^"
```

When executed, this line of code will display:

```
The house you are in has a master bedroom.
```

The **write** command above uses this macro by specifying the label of the location **bedroom** followed by **{list}**, the name of a macro. The other point of note with the above **write** command is the use of the caret character (^). The caret character is used to indicate where a new line on the screen should be started. The end of a **write** command does not automatically mean the end of a line of text.

The **{list}** macro is more commonly used with objects than locations, as only objects can be taken or placed inside other objects. It is more common for locations to be referred to using the **{the}** macro. This macro displays the object's definite article (the word **the** by default) followed by the object or locations **short** description text. For example:

```
write "You are in " bedroom{the} ".^"
```

will display:

```
You are in the master bedroom.
```

"Why not just write **You are in the master bedroom** directly?" I hear you ask. Given the above two lines of code, you could. The **{list}** and **{the}** macros are normally used, however, with an integer variable, not an object label. All objects and locations are assigned a unique number starting at one and numbered sequentially as they appear in your program. Using this numbering system, an integer variable can be used to refer to an object or location. The most common integer variable used as an object pointer is **noun1**. This variable represents the first object referred to in any given command typed by the player. For example:

```
write "There is nothing special about " noun1{the} ".^"
```

If the player was to type the command **examine bedroom**, the above code would produce the output:

```
There is nothing special about the master bedroom.
```

There is also one special purpose indefinite article: **name**. If **name** is specified as an object's article, the **short** description text will not be prefixed with anything whether displayed using **noun1{the}** or **noun1{list}**. This would normally be used when the description text is a proper noun. For more information on macros, see the section on [Printing the Names and Descriptions of Objects](#).

The JACL Author's Guide

The third line of our location definition states that if the player travels **west** from this location, then they should be moved to a yet-to-be-coded location with the label **bathroom**. You will learn more about this soon.

The next thing we will do is give the location its long description. This is the text that the player will see when they are in this location. We do this by associating a function called **look** with the **location** definition.

A function can be associated with an **object**, a **location** or be global. Functions are defined by starting a line with an opening curly brace ({) followed directly by the function's name. Any function whose name does not begin with a plus sign is automatically associated with the last **object** or **location** defined above it in the game file. Any function whose name does begin with a plus sign is said to be global. A global function is in no way associated with a particular **object** or **location**. A function definition is finished by placing a closing curly brace (}) on a line of its own. More information about functions and how they are associated with items is provided in the chapter on [Functions](#).

Below is the code for the **look** function to be added directly beneath the definition for the location **bedroom**. This function simply outputs plain text and therefore makes use of the **print** command instead of the **write** command we used earlier. This command takes no parameters and starts outputting all text from the following line until it reaches a line that has a period (.) as its first non-whitespace character.

```
{look
print:
    You are in your bedroom. There is a
    large, soft bed in the centre of the room
    while a doorway to the west leads into the
    bathroom.^
.
}
```

The colon after the print command is only an optional whitespace character used for presentation purposes. When using the **print** command, any following line of text that does not end with a caret (^), will have an implicit space added to the end before printing the following line. To prevent this space from being added, finish the line with a backslash (\) character. For more information see the chapter on [Screen Display](#).

As discussed earlier, the definition of the location **bedroom** refers to a second location called **bathroom**, located to its west. We will now add this second location to the game. The code for this new location should be placed after the **look** function that is associated with the bedroom:

```
location bathroom: bathroom
    short      the "bathroom"
    east       bedroom
    out        bedroom

{look
print:
    You are in the bathroom. The only
    exit from here is back east to the bedroom.
.
}

{movement
if compass = east : compass = out
    write "You bang your head as you walk "
    write "through the doorway.^"
    return false
endif
write "The only exit from here is to the "
```

```
write "east.^[^"
}
```

There are two points of note with this new location definition. Firstly, it demonstrates that it is quite valid (and in some cases highly desirable), to have more than one direction lead to the same location. In this case, if the player travels **east** or **out** from this location, they will be moved to the location with the label of **bedroom**. This completes the logical two-way connection of the bedroom to the bathroom, and the bathroom to the bedroom.

Secondly, it has a **movement** function associated with it. This function is called automatically by the interpreter whenever the player moves, or attempts to move, out of this location. If this function does not exist or returns **false**, the move will continue as normal. If the function returns **true**, the move will be prevented from taking place. Any function that reaches its closing brace will terminate as though it had issued a **return true** command. For move information on movement functions, see the section on [Movement](#). The **if** statement in the movement function says that if variable **compass** equals **east** or **out**, then the code up until the matching **endif** command should be executed. The variable **compass** is set to the direction the player is attempting to move in by the interpreter before the **movement** function is called. For more information on the **if** command, see the section on [Flow Control](#).



east and **out** are constants predefined in the interpreter to represent static numerical values. All the directions that can be travelled in from a location are stored in a 12 element array. These numerical values correspond to the index of the array element that stores the destination for that direction.

The Player

Now that we have our small, two-location world, we need a player to explore it. There is a definition for an object with the label **kryten** in **frame.jacl**. This object has been made with the specific purpose of representing the player and looks as follows:

```
object kryten: myself self me
short      name "yourself"
has        ANIMATE; FEMALE
capacity   42
parent     ;SPECIFY STARTING LOCATION HERE
player
```



The label **kryten** was chosen for the object representing the player after playing Infocom's Zork I with Frotz's **-o** option. This shows the object representing the player as having the name **cretin**. Being both an Infocom fan and a Red Dwarf fan, the choice was obvious.

The first line of this definition starts with the keyword **object**. This tells the interpreter that we want to define a new object with the label **kryten**. The words following this label are a list of names for the object. This, as you will have noticed, is the same format as defining a new location.



In JACL, a colon is treated as white space (as is a comma and tab). I simply use a colon in preference to a space in certain places to make the code more readable.

The second line contains a **short** keyword. This has the same purpose as the **short** keyword associated with bedroom above. In the case of our object **kryten**, however, the word **name** appears as the indefinite article.

The JACL Author's Guide

When using a **{list}** or **{the}** macro, anything other than the word **name** is printed verbatim. When **name** is specified, the JACL interpreter will not prefix the **short** description text with anything. For example, when **noun1** points to the object **kryten**, the following code:

```
write "You can't take " noun1{the} .^
write "You can't take " noun1{list} .^
```

will produce the output:

```
You can't take yourself.
You can't take yourself.
```

The third line of the object definition starts with the keyword **has** followed by the attribute **ANIMATE**. Both objects and locations may have as many or as few of the available attributes as required. Attributes are simply boolean flags that can be given to and taken from objects and locations at any stage during the game. They are frequently tested for by code in **verbs.library**. For example, the standard function to open an object will first check that it doesn't have the attribute **LOCKED** before taking away the attribute **CLOSED**, thus resulting in it becoming open. The attribute **ANIMATE** is tested for by functions associated with actions such as talking. If the player attempts to talk to an object that does not have the attribute **ANIMATE**, they are informed that the action is not logically possible. For a more information, see the chapter on [Attributes](#).

Following the attribute **ANIMATE**, you will see a semicolon then the attribute **FEMALE**. A semicolon (or hash symbol) signifies that all text following it, until the end of the line, is a comment. Comments have no effect on a game, they are only there to serve as notes for the author's benefit. In this case, deleting the semicolon will mean that the extra attribute will no longer be ignored. This will cause the player to be referred to in the feminine by all code in **verbs.library**.

The fourth line, beginning with the keyword **capacity**, indicates how many **mass** units the object can hold. In the case of an object with the attribute **ANIMATE** (such as this object representing the player), it indicates how much they can carry. In the case of an object with the attribute **CONTAINER**, it indicates how much can be placed inside it, while in the case of an object with the attribute **SURFACE**, it indicates how much can be placed on top of it.

The fifth line, beginning with the keyword **parent**, is followed by the label of another object or location. This indicates where the object is to be when the game starts. In this game, the player is to start in the location **bedroom**, so we must modify this line to read:

```
parent      bedroom
```

If an object doesn't have a **parent** property explicitly defined it will start the game with its **parent** property set to the nearest location defined above it in the game file.

The final line has the keyword **player**. This keyword tells the interpreter to set the interger variable **player** to point to the object **kryten** before the game begins. Only one object should have the keyword **player** defined, although the object pointer **player** can be changed to point to another object at any stage during the game if desired.

The properties that are associated with an **object** definition may be placed in any order. There are other properties that can be associated with an object that we have not set here, as the defaults used when they are absent are valid for our purposes. All of the possible properties are discussed in the chapter [Definitions in Detail](#).

Some Introductory Text

When a game is started or restarted, the function **+intro** is executed. The main purpose of this function is to display the game's title, the author's name and some text introducing the game. Any other commands required to set the initial state of the game-world can also be placed in **+intro**. Below is the **+intro** function for the tutorial game. This is simply the **+intro** function already present in your skeleton game file with an extra **print** block added after the title and the author of the game are written to the screen:

```
{+intro
style bold
write "^@" game_title
style normal
write " by " game_author "^^@"

print:
    Your alarm rings and you climb out
    of bed. Monday morning again so soon. Oh well,
    at least your house doesn't have a front door
    so you have a good excuse for not going to
    work.^@
.

if here hasnt OUTDOORS
    move north_wall to here
    move south_wall to here
    move east_wall to here
    move west_wall to here
endall

move ground to here

look
}
```

This function begins by using the **style** command to set the font to **bold** and then printing the value of the string constant **game_title**. This will be whatever string you set it to at the top of your game file. Next the font style is set back to **normal** and the value of the string constant **game_author** is output. The author's name is followed by three newline character to insert two blank lines. For more information on these commands see the chapter on [Screen Display](#).

After the title header, a game-specific print block outputs the introductory text for this game. This is the text that should set the scene for the player and introduce them to the game character they are playing.

The next block of code moves all the wall objects to the current location if the current location doesn't have the attribute **OUTDOORS**. These objects are defined in the file **frame.jacl** and should be moved to the current location each time the player moves to a location that doesn't have the attribute **OUTDOORS**. This is done using a similar block of code placed in the global function **+movement**. When the function **+intro** is executed, the current location will always be the starting location. For this reason, the **if** statement is not strictly necessary, but it is a good safeguard none the less.

The final line of the function contains a **look** command. This command prints the description of the current location and objects that are in this location.

Below is the default content of the function **+movement** that is executed each time the player attempts to move out of the current location:

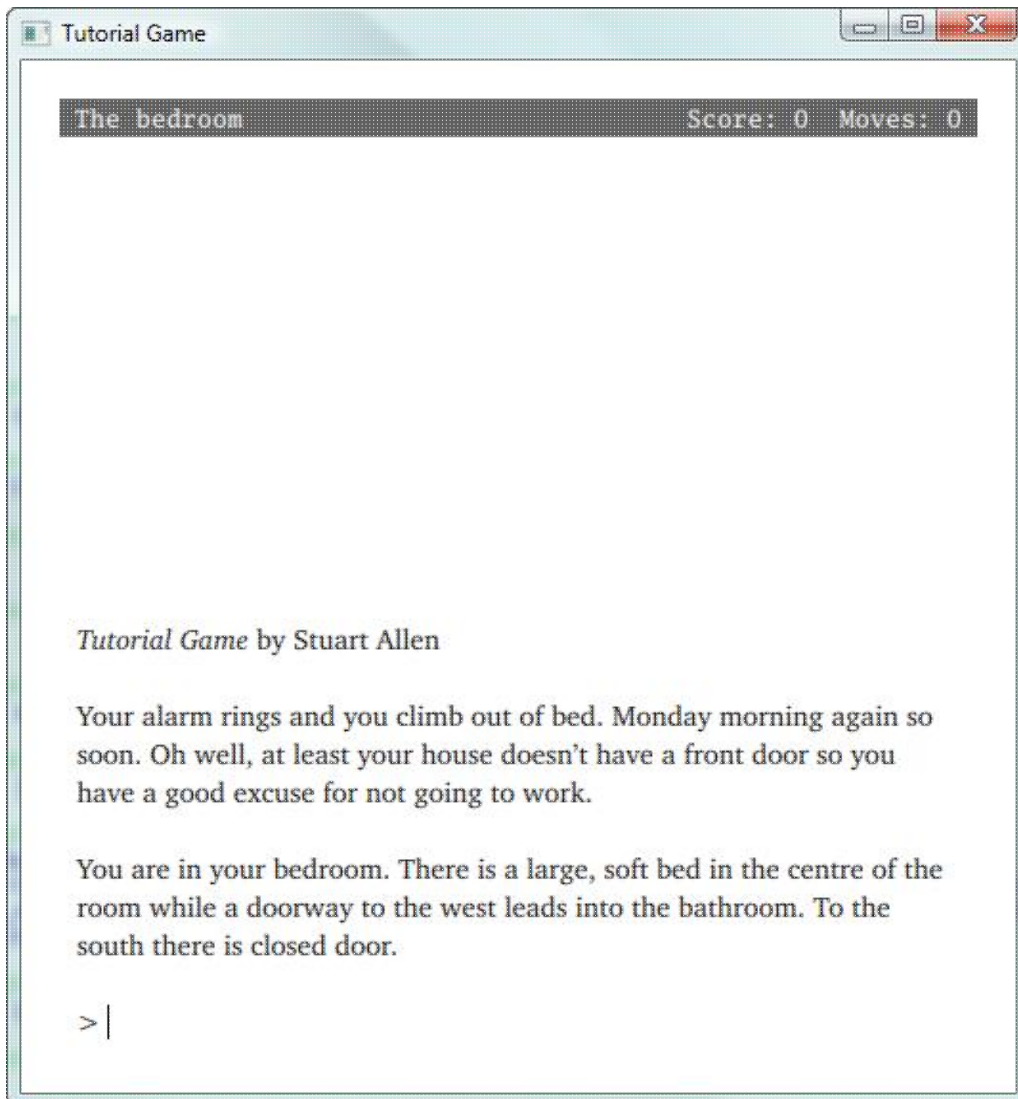
```
{+movement
ifall destination != nowhere : destination hasnt OUTDOORS
  move north_wall to destination
  move south_wall to destination
  move east_wall to destination
  move west_wall to destination
endif
return false
}
```

Objects

You can now try playing the beginnings of this game by first typing the command from within the **games** subdirectory:

```
../bin/garjacl game.jacl
```

When you do so, you should see something like the screen below:



The JACL Author's Guide

Once playing, the following commands should give you the responses shown:

```
>w
You are in the bathroom. The only exit from
here is back east into the bedroom.

>i
You are empty-handed.

>smell
Nothing strikes you as out of the ordinary.

>listen
You don't hear anything out of the ordinary.

>sit
You plonk yourself down for a moments rest.

>examine north wall
There is nothing special about the north wall.
```

As you can see, at this stage there is very little for the player to interact with in this game. Therefore, the next thing we will do is add another object: a small wooden box. We add the small wooden box by inserting the following object definition after the bathroom's **look** function:

```
object box: small wooden box
  has      CLOSABLE CONTAINER CLOSED
  short    a "small wooden box"
  long     "There is a small wooden box here."
  mass     25
  capacity 20
```

The first line says that we are defining an **object** and that it should have the label **box**. It then goes on to say that it can be referred to by the player with any combination of the names **small**, **wooden** and **box**.

The second line states that it should have the attributes **CLOSABLE**, **CONTAINER** and **CLOSED**. These attributes tell the appropriate verbs in the library that this object may be opened and closed, have things placed inside it and that it should be closed when the game begins.

The third line, as with the **short** statement for the object **kryten**, indicates the text to appear when either the object's **{list}** or **{the}** macro is used. In this case, however, rather than **name**, the article is set to **a**. Therefore, when **noun1** is set to **box**:

```
write noun1{list}
```

will display "a small wooden box", while

```
write noun1{the}
```

will display "the small wooden box".

The fourth line, beginning with the keyword **long**, details the text to be displayed when this object is in the current location. If an object has a **mass** of **scenery**, the **long** text will not be displayed. This is because a **mass** of **scenery** indicates that the object cannot be taken and should therefore be described in its parent location's **look** function instead, if at all.



Internally, **scenery** is a constant with a value of 100 while **heavy** has a value of 99. An object that has its **mass** set to **heavy** will have its **long** text displayed when the object is in the current location, but can't be taken by the player. It is good to keep this in mind when choosing a **mass** value for each object.

The fifth line sets this object to have a **mass** of 25. When the player takes an object, the value of the object's **mass** property is subtracted from the player's **capacity** property. This indicates that the player's **capacity** property must currently be 25 or greater for him or her to be able to take the box. Any other container object the player attempts to put the box in or on must also have a **capacity** property that is currently 25 or greater.

The final line, a **capacity** property, indicates how many **mass** units the box can hold. In this case it is set to 20. This means that other objects may be placed inside this object (due to it having the **CONTAINER** attribute) until the total of their **mass** properties equals 20.

As this new object has no **parent** property, it will start the game in the bathroom. This is because the bathroom is the nearest location defined above it in the game file. Any objects with no **parent** property will begin in the nearest location above it regardless of the number of object or function definitions inbetween.

Before we play the game again, we will add a second object by putting the following code beneath the definition for the box.

```
object note: orange note
  short    an "orange note"
  long     "An orange note rests on the ground."
  parent   box
  mass     5
```

As you can see, the note's **parent** property is followed by the label of the box object. This indicates that the note is to start the game inside the box. If this **parent** property were to be omitted, the note would begin in the bathroom along with the box, as this is the last location defined above it in the game file.

If the box object had the attribute **SURFACE** rather than **CONTAINER**, the note would start the game on top of it. If the box object had the attribute **ANIMATE**, the note would start the game being carried by it.

Verbs and Functions

Now if you restart the game, you can walk west into the bathroom and find the small wooden box. You can take it, open it, look in it, close it, and apply a whole range of other standard verbs present in **verbs.library** to it. When it is open, you will also be able to take the orange note out of it.

To explain a little bit about how verbs work, let's take a closer look at the verb **read**. The **grammar** definition and the global function for the verb **read** can be found in **verbs.library** and are reproduced here:

```
grammar read *present >read

{+read
if +important<noun1 = true
  return true
endif
if +darkness = true
  return true
endif
write "There is nothing on " noun1{the} " to read.^"
}
```


This **grammar** definition states that if the player types a command of the format "**read** *ObjectThatIsPresent*" during the game, then certain functions with the base name **read** should be executed.

When the player makes a move, the JACL interpreter will set the integer variables **noun1** and **noun2** to point to the objects referred to in the command. They will be set based on the order the objects appear in the move. For example, if the player types the command **give sword to troll** the **+give_to** function would be called with **noun1** being set to the sword and **noun2** being set to the troll.

Returning to our read example, upon identifying the player's command as matching this **grammar** definition, the JACL interpreter will first attempt to execute a function called **read** that is associated with the object that the player is attempting to read. If this does not exist, it will try to execute the same function name only prefixed with a plus sign. This is a global function and can be thought of as the default action that occurs for that verb if no object-specific one is provided.

Since we have not associated a **read** function with our **note** object, when the player attempts to read it the global function **+read** will be executed. If you try this you will see that the default action for the read verb is very simple, but appropriate for most objects. The default function **+read** is not an appropriate, however, for our note object. We will therefore replace it by adding a function called **read** that is associated with the object **note**. This function is even simpler:

```
{read
write "Welcome to Jamaica and have a nice day.^"
}
```

We associate this function with the note object by typing it directly after the note object's definition. For clarity, you can of course leave a blank line or two in between the note definition and read function. This local **read** function will now be executed in place of **+read** whenever the player types the command **read note**, provided the note is visible to the player.



The **read** function above should really be given two names. This is done by placing the second name after the first, such as **{read : examine}**. This means that both reading and examining the note will lead to the same code being executed. Any function may have as many names as can fit in a single line of JACL code.

Overriding Functions

Compared to **+read**, the **+close** function is quite long. Although the default **+close** function is perfectly suitable for our needs, we will use it to demonstrate overriding the default outcome of a function. The **+close** function is reproduced here:

```
{+close
if +important<noun1 = true
  return true
endif
if +darkness = true
  return true
endif
if +reach<noun1 = true
  return true
endif
if noun1 hasnt CLOSABLE
  write "You can't close " noun1{the} .^
  set time = false
  return
}
```

```
endif
if noun1 has CLOSED
  write noun1{The} noun1{is} " already closed.^"
  set time = false
  return
endif
override
write "You close " noun1{the} .^
ensure noun1 has CLOSED
}
```

During the course of the **+close** function, several tests are performed to determine the appropriate outcome. At the point in the function where it is decided that the command should be successful there is an **override** command. This command tells the JACL interpreter to look for a function called **close_override** that is associated with the object that the player is attempting to close. If this exists it will be executed in place of anything beyond the **override** command. If this does not exist, then execution of the **+close** function continues as normal from the line following the **override** command.

The reason for the **override** command is that a **close** function that is associated with any object will get called straight away, completely replacing all the code in the default function **+close**. This means that any test, such as whether the object is already closed, will have to be repeated manually in the new, local function. This is not so much of a problem with a simple verb like **read**, as no tests are performed, but with some other verbs this can be a considerable amount of code. The **override** command therefore provides an opportunity to override only the outcome, not the entire function.

To demonstrate, we will override the normal outcome of closing the box. This is done by associating the following function with the object **box**:

```
{close_override
write "The lid creaks as you push it closed.^"
ensure box has CLOSED
}
```

When the box is closed by the player, all the tests before the **override** command in the **+close** function will be executed. If all the tests pass, the code in **close_override** will be executed in place of any code after the **override** command.



It is important to be careful that you use an override function (or perform the tests manually) whenever a verb has several possible outcomes. If the above function was to be called **close**, as opposed to **close_override**, then the box could be closed over and over again - clearly a bug.

Doors

We will now add a third and final location. This location will be a living room and will be placed south of the bedroom. We are also going to place a door between the bedroom and living room. Doors are common in interactive fiction, implemented using a regular object, and yet in some ways odd and out of the ordinary. The thing that makes a doors different to most of the other objects in your game is that they live between two locations, rather than inside one, and need to be accessible from both locations.

Before we move onto the door itself, add the following location definition beneath the **close_override** function that is associated with the box:

The JACL Author's Guide

```
location living_room: living room
  short      the "living room"
  north      nowhere

{look
if here has VISITED
  print:
    You have returned to the living
    room.^
.
else
  print:
    You are in the living room. There
    is a small television perched on a low-lying
    table in front of a sofa.^
.
endif
}
```

The **look** function associated with the living room is slightly more complex than those associated with the other locations. After the player enters a location for the first time, it is automatically given the attribute **VISITED** by the JACL interpreter. This latest **look** function tests the current location for this attribute and displays a shorter, more appropriate description if it has it. Feel free at this stage to go back and add this test to the **look** functions for the bedroom and bathroom. The **north** direction is specified as leading to **nowhere** as the game will start with the door closed. As **nowhere** is the default for an unspecified direction, we could have simply omitted it. I prefer to specify **nowhere** for directions that only *currently* lead nowhere, but will change during the course of the game.

As you may have guessed, I would also add this line to the bedroom giving the complete definition of:

```
location bedroom: master bedroom
  west      bathroom
  south     nowhere
```

As you will see later, we will make the door accessible from two locations by moving it around, so it does not really matter when you define its object. As the player begins the game in the bedroom, and this is where they will first encounter the door, beneath the bedroom definition is as good a place as any for the door's definition:

```
object door : bedroom door
  short      the "bedroom door"
  has        CLOSABLE CLOSED

{open_override
set bedroom(south) = living_room
set living_room(north) = bedroom
return false
}

{close_override
set bedroom(south) = nowhere
set living_room(north) = nowhere
return false
}
```

The above code defines an object for the door that has the attributes **CLOSABLE** and **CLOSED**. These attributes allow the verbs in the library to manipulate the door appropriately. Like the box, the outcome of the **close** verb is overridden, as is the outcome of the **open** verb. Unlike the box, the override functions for the door

both end with a **return false** command.

As discussed above, when the interpreter encounters the **override** command in the global function **+close**, it will look for the function **close_override** that is associated with the object being closed. If this function does not exist, or returns **false**, execution will continue from the first line after the **override** command. As we have defined an associated **close_override** function, it will be executed at this point. It is possible, however, for the override function to issue a **return false** after a test has determined that the default outcome is sufficient in a particular instance. If this happens, the **close_override** function simply provides some code to execute *in addition* to the default outcome. In other words, the default outcome will occur as if the override function did not exist at all if the override function returns **false**.

Now it is time to move the door between the bedroom and living room so that it is accessible from both locations. As with most programming challenges, there are many ways the desired effect could be achieved. One option is to create two doors, one in the bedroom and one in the living room, then keeping these two doors synchronised. In this case we are going to add two **eachturn** functions: one associated with the bedroom and one with the living room. These two functions move the door to the current location and each have a single line of identical code:

```
{eachturn
move door to here
}
```

It is possible to give a function more than one name by supplying a list of names separated by white space. This is a very common technique for making several possible moves by the player cause the same outcome. For example, in *The Unholy Grail*, the message for examining the rod for opening and closing the blinds is exactly the same as the one for attempting to take it:

```
{examine : take
print:
  The plastic rod is attached to the blinds and
  can be turned in order to open and close them.^
.
}
```

Internally, when a function is associated with an object, the function's name is stored as the supplied name followed by an underscore, then the label of the object it is associated with. This means that the above function would have two full internal names being **examine_rod** and **take_rod**. As the two **eachturn** functions in this tutorial game have identical content it would be good to use a similar technique to avoid the duplication. The difference between the **eachturn** functions and the **examine** and **take** functions is that with the **eachturn** functions, it is the same action for two different objects (in this case, locations) rather than two different actions for the same object. It is possible to overcome this problem by prefixing one of the function's names with an asterisk (*), and then supplying a full internal name manually. The asterisk will not become part of the full internal name (unlike the plus sign at the start of a global function), it simply tells the interpreter that you are going to handle the association manually and that it should not automatically add the label of the object above it in the source file as a suffix.

Using this technique we are able to write a single **eachturn** function and associate it with both the bedroom and the living room by adding the following code somewhere below the definition for the living room:

```
{eachturn : *eachturn_bedroom
move door to here
}
```

This will create a function that has two names. The full internal version of these two names will be **eachturn_living_room** and **eachturn_bedroom**. The first association is done automatically while the second is done manually by using the asterisk prefix. Don't worry if this doesn't make complete sense to you right now, it is not an essential technique for writing JACL games, particularly with functions as short as this **eachturn** function. More information on creating a function name using the asterisk prefix can be found in the chapter on [Functions](#).

Regardless of whether you use this technique, or associate an identical **eachturn** function with the bedroom and the living room in the usual manner, you will now have a functioning door. The door is handled correctly by the library verbs **open** and **close** due to having the **CLOSABLE** attribute and it is always accessible from the bedroom *and* living room thanks to the **eachturn** functions. Finally, the extra code supplied in the **open_override** and **close_override** functions create and destroy the extra links between the bedroom and living room as appropriate.



Information

It is also possible to have moved the door from location to location using **movement** functions that are associated with bedroom and living room. These functions would need to test which way the player is travelling, and move the door to that location if it is into the bedroom or living room. Although this requires slightly more complicated code, it is more efficient as the move is only done once when required, not every turn that the player then spends in that location. With a small game like this, however, simplicity is of greater concern than performance.

Non-player Characters

Most games will include at least one character other than the player, and this small tutorial game is no exception. The character we will add is the player's son who will be sitting in the living room watching television. Fortunately for us, simulating the responsiveness of a teenager watching television is not hard.

We will begin by adding the following two object definitions and their associated **examine** functions beneath the definition for the living room:

```
attribute EXAMINED

object television: television tv tele
  short      a "television"
  mass       scenery

{examine
if self has EXAMINED
  write "It's Rick who is the TV addict, not you.^"
  return
endif
write "There is currently a cartoon showing on the "
write "television.^"
ensure self has EXAMINED
}

object rick: son boy teenager rick
  has        ANIMATE
  short      name "Rick"
  long       "Rick is here, watching television."
  mass       heavy

{examine
if @ = 1
  print:
    Rick is staring blankly at the television screen.^
.
else
  print:
    Rick is still gazing into the television's screen.^
.
endif
}
```

Before the object definitions is the definition of a user attribute called **EXAMINED**. Up to 32 user attributes can be defined, and like the system attributes **CLOSABLE** and **CLOSED**, they don't have a value, they are simply flags that an object either has or hasn't. The **examine** function for the television makes use of this attribute to test if the television has already been examined before. Unlike defining a constant as a synonym for one of an object's integer properties (see the section on The Passing of Time below), user attributes are not synonyms for system attributes, they are an entirely separate set of 32 attributes.

The **examine** function for Rick achieves a similar result using an entirely different mechanism. When the JACL interpreter executes a function it automatically creates and increments a counter that records how many times that function has been called. This counter is referred to in code as an at sign (@) followed by the name of the function. For example, the number of times the function called when examining Rick has been called can be checked from any other code by examining the contents of the container **@examine_rick**. When an at sign is referred to on its own, such as in the example above, it is taken to refer to the call count of function that

is currently running.

To make both Rick and the television respond to the player's commands, we will be required to associate more functions with each of them. This process of creating objects and associating functions with them is the essence of writing text adventure games using JACL.

The first action we will cater for is talking to Rick. The **grammar** statement that matches the command **talk to rick** calls the function **talk_to**. Therefore, in order to give a custom response to this command, we must associate a **talk_to** function with the object **rick**. This function should look as follows:

```
{talk_to
print:
  ~Uh, yeah, I'll do it in a minute,~
  Rick mumbles with out looking up. You have
  quite a strong suspicion that he didn't
  really hear a word you said.^
.
}
```

The above **talk_to** function is associated with Rick in the same way that the **read** function was associated with the note earlier. For a complete list of all the grammar statements defined in the library and the names of the functions they call, see [APPENDIX B: Library Verb Functions](#).



As double quotes are used to enclose any command parameter that contains spaces, attempting to print a double quote directly to the screen will not give the desired results. Wherever you want to print a double quote put a tilde character (~) and the interpreter will print a double quote in its place.

To code a response to the player asking Rick about something in particular, you need to associate a function that has a compound name with the object **rick**. The **grammar** definition for the command **ask noun1 about noun2** calls the function **ask_about**, so this is the beginning of the function name. You specify which object is being asked about by appending an underscore and then the label of the object to the function name. For example, the following function (when associated with the object **rick**) will be called if the player types the command **ask rick about note**:

```
object rick: son boy teenager rick
...

{ask_about_note
print:
  ~I don't know nothing about no note,~ Rick says
  looking at your blankly.^
.
}
```

This gives a custom response when the player asks Rick about the note, but what if we want to give the same custom response no matter what object Rick is asked about? The hard way would be to associate a function for each and every object that prints your required response. The easy way is to write a **+default_ask_about** function and test whether the person being asked is Rick. Any **+default** function can be thought of as a replacement for the code that comes after the **override** command. If you search through the **verbs.library** for **override** commands you will see that they occur at the point in a verb's global function where it has been decided that the verb should be successful. This allows a **+default** function to override the default outcome of a verb without needing to repeat all the preliminary checks. The global **default** function for any given action is called when an **override** command is reached and a specific override function for the object or objects in

question does not exist. The order that functions are called in and the precedence they have over each other is fully detailed in the chapter on [Functions](#).

This is **+default_ask_about** function is a global function (as indicated by the leading plus sign), so it doesn't matter where you put it in the code. Here is the code to create a default response for whenever Rick is asked about an object:

```
{+default_ask_about
if noun1 = rick
  print:
    Rick blinks several times then
    pokes out his bottom lip. This, you have
    figured out over the years, translates to,
    ~Not a clue.~^
.
  return
endif
return false
}
```



If both this function and the specific **ask_about_note** function exist, the **ask_about_note** function will be executed if player types **ask rick about note** in preference over this function.

The first thing this function does is test the current value of **noun1**. As discussed above, **noun1** and **noun2** are object pointers that point to the first and second objects referred to in the player's move. If this is currently set to **rick**, our new default message will be displayed. If **noun1** is not set to **rick**, a **return false** command is executed. This command causes the interpreter to do whatever it would have normally done had this function not existed at all. In this case, the result would be to perform the original default action specified in the library.

The Passing of Time

In many games actions will occur based on the passing of time rather than the direct actions of the player. If the player was to do nothing other than type **wait**, people would still come and go, the song playing on a nearby radio would change and the sun would set. All these events would be coded for within an **eachturn** function. There can be an **eachturn** function associated with each location and a single global one.

The variable **time** is automatically set to **true** before the player types each of their moves. If the player's move is not possible, this variable should be set to **false**. If, when the command has been fully processed, **time** is still set to **true**, the appropriate **eachturn** functions are executed. The first function executed, if it exists, is the **eachturn** function that is associated with the location the player is currently in. When this has finished, the global function **+eachturn** is executed. Once **+eachturn** has finished executing, the processing of the player's command is complete.

To demonstrate, we will make Rick take a sip from his drink every five turns, regardless of what the player does. As this would happen regardless of whether the player is in the room or not, we will put the code to handle this in the global **eachturn** function:

```
constant turns_since_last_sip      5

{+eachturn
set rick(turns_since_last_sip) + 1
```



```
if rick(turns_since_last_sip) = 5
  if here = living_room
    write "Rick takes a sip from his drink.^"
  endif
  set rick(turns_since_last_sip) = 0
endif
}
```

Each object has sixteen integer properties that can be set and tested. The properties **parent**, **capacity** and **mass** are three of these integer properties that you have already seen. For a complete listing, see the chapter [Definitions in Detail](#). These sixteen properties are stored in an array, with **parent**, **capacity** and **mass** being the first three integers, stored in the order starting at index 0. This means that the following code will print the number 25 twice:

```
object cube : timber cube
  short  a "timber cube"
  mass           25

{+some_function
...
write cube(mass) ^
write cube(2) ^
...
}
```

As the **parent** property is an integer property, only the index of the parent object is stored. When objects are defined in your game, there are given a sequential integer index starting with 1. As objects and location are both stored internally as objects, this numbering system makes no distinction between the two. This means that if a location representing a forest is the first object or location defined in the game, setting any other objects **parent** property to 1 will place it in the forest. Although potentially dangerous if you rearrange your game code, this also allows you to test if the player is in a certain section of your game world by testing if the value of **here** (an internally defined synonym for **player(parent)**) is within a certain range of values.

The **status** property of an object is set and tested in the same way as the **parent** property. The **status** property, however, has no pre-determined use so we are free to use it as our counter. In order to make our code more readable and self-commenting, before the **+eachturn** definition, is a constant definition with the name **turns_since_last_sip** with a value of 5. The **status** property is the sixth integer property and therefore has an index of 5. By creating this constant we are able to use an object's **status** property through a more appropriate synonym. The code in the **+eachturn** function increments this property by one after each successful move made. When it equals 5 a message is displayed (if the player is in the living room) and it is set back to zero. The process will then loop over and over again.

Winning and Losing the Game

Our mini game is not much good unless it can be won. For this to happen, the player must have a goal and be awarded points for each obstacle they overcome along the way. The goal for this game is going to be to find the television guide and give it to Rick. To make this game a fiendishly-clever all-time classic we are going to hide the guide under the bed.

Before we do this, however, we are going to introduce an element of danger by adding some code to cater for switching the television off. We already have an object for the television, so all we need to do now is associate a **turn_off** function with it. This should look like the following:

```
{turn_off
```

The JACL Author's Guide

```
print:
  As you reach over and switch off the
  television, you get quite a shock to see Rick
  rapidly growing a coat of hair and foaming at
  the mouth. The shock of this is only surpassed
  by that of him sinking his newly
  acquired fangs into your throat.^
.
execute "+game_over"
}
```

Okay, so killing the player without any real warning is grossly unfair, but it serves as a demonstration of how to handle the player dying. The last line of the above function is an **execute** command that calls another function with the name **+game_over**. This function is included in the file **verbs.library** and looks like this:

```
{+game_over
write "^"
execute "+score"
loop
  if noun3(parent) = player
    set noun3(parent) = limbo
  endif
endloop
set player(parent) = prologue
write "^"
execute "+look_around"
}
```

This function uses two other **execute** commands to call two other functions: one to display the player's score and another that displays the contents of the location **prologue**.

Before we move on to winning the game, it is worth mentioning that the default response for turning an object on is to say that this cannot be done. In the case of the television, it would be important to add a **turn_on** function stating that the television is already on.

Now, on to the television guide. To implement this puzzle we are going to need two more objects: the guide itself and the bed to hide it under. Begin by defining an object for the bed somewhere beneath the definition for the bedroom (but before that for the bathroom). Beneath that, add a definition for the television guide. The television guide must have its parent property set to **limbo** until the player has discovered it. The location **limbo** is defined in the library and is used exclusively for situations such as this where we need somewhere to temporarily store objects that the player should not have access to. Finally, we must associate a **look_under** function with the bed that moves the guide to the bedroom and awards the appropriate points. Here is the complete code for all of this:

```
object bed: bed
  short      a "bed"
  mass       scenery

{look_under
if guide(parent) = limbo
  print:
    Hidden under the bed you
    find this week's television guide.^
.
  set guide(parent) = here
  points 50
  return
```

The JACL Author's Guide

```
endif
write "You don't find anything else.^"
}

object guide: television tv tele guide
  short      a "television guide"
  long       "The television guide is here."
  parent     limbo
  mass       5

{examine : read : look_in
write "It contains a listing of this "
write "week's programmes.^"
}
```



It is important with objects that share names, such as the television and the television guide to be aware of which object has the most names. If only shared names are used, the object with the lowest number of names will be selected. For more information, see the chapter on [Object Resolution](#).

Now, when the player looks under the bed, the guide will be moved from its initial location, **limbo**, to the bedroom. Once this has been done, the player will be able to take it. The **points** command will increase the player's score by 50%. The **if** statement in this function ensures that this can only be done once.

We will now associate a **give_to_rick** function with the guide. This will be the winning move and should look like this:

```
{give_to_rick
print:
  ~Cool!~ Rick exclaims as he
  snatches the guide from your hands.^
  Satisfied that you have achieved
  at least one thing today, you decide to
  go back to bed.^
.
points 50
execute +game_over
}
```

And so the game is won. The extra 50 points give the player a total of 100, and the function **+game_over** is executed.

In order to make this tutorial game more complete, other moves the player is likely to try would need to have custom responses added by associating the appropriate functions with the appropriate objects. The more obvious of these include: showing the guide to Rick, telling Rick about the guide and sleeping on the bed. In fact, the more moves you can give custom responses to, the more depth and character the game will have. Also, anything prominent that is mentioned in a location's description, such as the table and sofa in the living room, should also be defined as objects. This enables the player to refer to them, even if they aren't important to solving the game.

If you have an object that logically needs to be in your game as a part of the scenery, but don't want players to waste their time on it as it has no real purpose, you can give it the attribute **NOT_IMPORTANT**. This attribute causes all the verbs in the library to give a special response that tells the player there is no need to worry about interacting with this object. By giving a minimal implementation of objects this way, you avoid the player receiving a "You can't see any such thing." message, without the object being perceived as a red herring. For example, this could be used with the sofa in living room using the following minimal code:

The JACL Author's Guide

```
object sofa : sofa
  short      a "sofa"
  has        NOT_IMPORTANT
```

No anytime the player attempts to refer to the sofa they will receive the message:

```
The sofa is not important, you don't need to worry about that.
```

Although not a large or complex game, this tutorial game does demonstrate most of the elements of JACL that you will need to create a more complete piece of interactive fiction. The rest of this guide contains a complete reference description of every feature of the JACL language and interpreters. It is recommended that you at least skim through these chapters in order to gain an awareness of the features at your disposal.

Testing, Debugging and Releasing

After you have finished writing your game it is important to test it thoroughly before it is released. In a game of any real size, many bugs exist and will only be discovered through extensive beta testing. This chapter covers some tools and techniques for debugging a piece of interactive fiction written using JACL.

The most immediate indication you will get of a problem with your game is an error message while trying to load the game. Errors with the definition of data (such as objects and variables) that are detected while loading your game will specify the line number the error occurred on. The line numbers specified are those from the **.j2** file created in the **temp** directory. If the only files you include in your game (using the **#include** directive) are at the very end, the line numbers specified in the error messages will be equivalent to the lines in your original **.jacl** source file. If you include files at various points before the end of your game, the line numbers specified will not be equivalent. In order to investigate the offending line of code in this situation you will need to look directly at the processed **.j2** file.

The WALKTHRU Command

The **walkthru** command allows you to specify the name of a text file that contains a list of moves to process as though they were typed by the player. The text file must contain one command per line and may contain as many commands as you would like. The default name for a walkthrough file is *GameName.walkthru*. It is common for this default file to issue all the commands required to get from the beginning of the game to end. This can vary from the most direct path using the minimum commands to the most thorough path visiting every part of the game along the way. Other shorter walkthrough files may also be created that take the player from one specific point in the game to another.

Walkthru files have two advantages over saved game files. The first is that as soon as you add or remove an object or variable to a game, saved game files made using a previous version of the game can no longer be loaded. This means that without the use of a walkthrough file, play testing the end of the game would involve tediously playing the beginning of the game over and over again as it grows. The second advantage is that a walkthrough file shows all the output from every command rather than just instantly teleporting you to a future (or past) point in the game. This makes walkthrough files useful for regression testing. By running through a walkthrough file after changing some code in your game, you are able to test that everything that previously works still works as expected by quickly scanning the transcript.

When a walkthrough script is running, no **[MORE]** prompts will be displayed, including those explicitly displayed by a **more** command. Below is a sample of a walkthrough file, being the top ten lines from **grail.walkthru**:

```
w
n
turn off lights
close blinds
examine screen
s
s
take gps
n
w
```

As you can see, **walkthru** files are of a simple format that can be created with any text editor.

Transcripts

The JACL interpreter provides a mechanism for recording a transcript of a game while it is being played. Recording is started by using the **script** command then entering a filename to save the transcript to.

As well as being interesting records to keep of your playing, transcripts are an enormously useful way of receiving feedback from people who beta test your games. While recording a transcript it is possible to insert a comment by typing an asterisk as the first character at the command prompt then following it with the comment you would like to add. For example, the in-game command:

```
>*It is possible to open the door even when it is locked.
```

will appear in the currently active transcript with only the following response:

```
Comment ignored.
```

You stop a transcript from being recorded by using the command **unscript**.

The Debug Library

The library **debug.library** contains a set of JACL verbs that are useful for testing your game during development. This library should be included using the **#debug** directive so that it is not included in the release version of your game (see below).

The INSPECT Command

The **inspect** command displays the label, attribute and element values of the object passed as a parameter. Two special verbs in **debug.library** allows an object to be inspected while playing a game, these are **inspect** and **inspectobj**. Below is some sample output of the **inspect** verb from *The Unholy Grail*:

```
>inspect drawer
label: compartment
has object attributes: CLOSED CLOSABLE CONTAINER
has user attributes:
parent: device (112)
capacity: 6
mass: 100
bearing: 0
velocity: 0
next: 0
previous: 0
child: 0
index: 0
status: 0
counter: 0
points: 0
class: 0
x: 0
y: 0
```

The above output is a dump of the current values of all of the object's properties, including attributes. The object chosen by the **inspect** verb is determined using the standard rules for object resolution based on the names provided. In this case, the single name **drawer** has uniquely identified an object with the label **compartment**.

It is also possible to use the command **inspectobj** to give the same details as above for whichever object an expression resolves to. For example, the follow two commands will show the detail of the player's current location:

```
>inspectobj player(parent)
>inspectobj here
```

The Verb VALUEOF

The **valueof** verb is a way of displaying the current integer value of any resolveable container. This includes an object label, an object element and an integer variable or constant. Below is an example of the **valueof** command being used while playing The Unholy Grail:

```
>valueof report(parent)
report(parent) = 44
```

The Verb **FETCH**

Where the verb **inspect** is a way of accessing the JACL command **inspect** during game play, **fetch** is simply a convenience verb to move an object to the current location and remove the attribute **OUT_OF_REACH** from that object should it have it. The verb is handy when you need to test a puzzle that involves the use or one or move objects that are time consuming to obtain manually.

For thorough testing it is preferable to use the **walkthru** verb to test all the steps normally taken to obtain the object or objects, but **fetch** can still be of use when testing small parts of your game in isolation.

Releasing Your Game

Once you have completed your game and all known bugs are fixed you will want to release it to the public to be played. Although it is possible to simply distribute the **.jacl** file you have been working on (presuming only standard JACL libraries are used), it is often preferable to distribute the **.j2** file. This file is created in the **temp** subdirectory when the game is run. This file can be thought of a statically-linked version of your game, with all **#included** files appended to the game file. In addition to appending all the required library files, unnecessary whitespace is removed from the beginning of each line to reduce file size and improve run-time performance.

Although a **.j2** file is created each time your game is run, it is possible to use the **-release** argument when running **jacl** to produce a file more suitable for distribution. The argument will cause the **.j2** to be encrypted (mildly), and to only contain files included with the **#include** directive, not the **#debug** directive.

If you wish to create a **.j2** file that does not include the **#debug** files, but is not encrypted, you can supply the arguments **-release -noencrypt**.

Screen Display

Interactive fiction is primarily a text-based medium and this chapter discusses all the features of JACL that relate to the output of text. Depending on the Glk library used, JACL interpreters may also provide the ability to add graphics and sound to your games. The use of these features is covered in the following chapter, [Glk and Multimedia](#).

The WRITE Command

The **write** command is the most flexible method of outputting text. It takes one or more parameters, each separated by a whitespace, and prints the result to the screen. A parameter can either be plain text, a string or integer variable, a string or integer constant or an object macro.

For example, the following command:

```
write Hello, world!
```

contains two separate parameters, the string **Hello** and the string **world!** (the comma after **hello** is considered to be white space.) When executed it will produce the following output:

```
Helloworld!
```

This is obviously not the desired result. The correct way to print this familiar greeting is:

```
write "Hello, world!"
```

The reason for this demonstration of how not to print text is that it demonstrates how the **write** command operates. Each of the parameters supplied to the **write** command is printed directly after the one before it. In the second example a single parameter containing both a space and comma is printed.



As with all other JACL commands, any single parameter that contains spaces must be enclosed in quotes. Failing to do this is a common cause of error.

Special Characters

When printing text, the **write** command recognises the following special characters:

Character	Output
<code>^</code>	A caret will be translated into a newline.
<code>~</code>	A tilde will be translated into a double quote.

To print the first two special characters (`^` and `~`) literally, the words **caret** and **tilde** should be used as parameters of a **write** command.

Below is an example of the use of the above special characters:

```
write "~Hello,~ said the boy.^"
write "~Hello,~ said the girl in reply.^"
```

Each of these **write** commands prints a single parameter enclosed in quotes and together will display:

```
"Hello," said the boy.
"Hello," said the girl in reply.
```

Printing the Value of Variables

If the value of a variable is to be printed, as opposed to verbatim text, the name of the variable must be entered as a separate parameter that is not enclosed in double quotes. For example, consider the following line from the **+score** function of the standard library, a **write** command with five parameters:

```
write "Your score is " score "% in " total_moves " moves.^"
```

The output of this five parameter command will vary depending on the current value of the two variables being printed. Typed as the very first command of the game it would display:

```
Your score is 0% in 0 moves.
```

Printing the Value of Item Elements

The current value of object and location elements may also be printed using a **write** command. This is done by enclosing the name of the element in brackets directly after the object or location label. For example:

```
write "The dial is set to " dial(status) "Mhz.^"
```

The elements of each object and location are stored in a sixteen element array (0-15). A number between 0 and 15 can be used between the brackets as an index to the object element rather than it's name. For a mapping of default element names to index numbers, see the chapter [Definitions in Detail](#). Constants can also be defined as a way of renaming any given element to a name that more clearly indicates its purpose. For example:

```
constant fuel_left      2

{+show_status
set submarine(fuel_left) = 42
```

```
write "FUEL: " submarine(fuel_left) ^
}
```

This code defines **fuel_left** as the constant 2, then sets the second element of the object **submarine** to equal 42. Finally, it uses a **write** statement to output the current value of **submarine(2)** by using the constant **fuel_left** to improve the readability of the code. This results in the following output:

```
FUEL: 42
```

Printing the Names and Descriptions of Objects

As you can see in the file **verbs.library**, it is a common need to print the short description of the object or objects that the player referred to in his or her last move. This is done using the integer variables **noun1** and **noun2**. When used as parameters of a **write** command, these variables may be followed by either a **{the}** or a **{list}** macro to display the object's short description. Given the following object:

```
object wooden_box : small wooden box
  has          CONTAINER CLOSED CLOSABLE
  short        a "small wooden box"
```

if the player refers to this box as the first object in their command, then the **write** command:

```
write noun1{list}
```

would display, "a small wooden box", while

```
write noun1{the}
```

would display, "the small wooden box".

If you require the output to be capitalised when using these macros, use **{The}** or **{List}** instead. For example, with **noun1** still set to the box object:

```
write noun1{The} " is no good to eat.^^"
```

will display, "The small wooden box is no good to eat."

Sentences Referring to Varying Objects

There are several other macros that may be used with the **write** command to assist in constructing sentences that refer to different objects at different times. They are designed to simplify the process of displaying the correct text for objects based on whether they have the attributes **PLURAL**, **ANIMATE** or **FEMALE**. They are:

Macro	Output
<i>object_label</i> { long }	Prints the objects long description.
<i>object_label</i> { that }	Prints either the word them if the object has the attribute PLURAL , him if the object has the attribute ANIMATE , her if the object has the attributes ANIMATE and FEMALE , or the word that if the object doesn't have any of these

	attributes.
<i>object_label{it}</i>	Prints either the word they if the object has the attribute PLURAL , he if the object has the attribute ANIMATE , she if the object has the attributes ANIMATE and FEMALE , or the word it if the object doesn't have any of these attributes.
<i>object_label{does}</i>	Prints the word do if the object has the attribute PLURAL , or the word does if the object doesn't have the attribute PLURAL .
<i>object_label{doesnt}</i>	Prints the word don't if the object has the attribute PLURAL , or the word doesn't if the object doesn't have the attribute PLURAL .
<i>object_label{is}</i>	Prints the word are if the object has the attribute PLURAL , or the word is if the object doesn't have the attribute PLURAL .
<i>object_label{isnt}</i>	Prints the word aren't if the object has the attribute PLURAL , or the word isn't if the object doesn't have the attribute PLURAL .
<i>object_label{s}</i>	Prints the letter s if the object hasn't got the attribute PLURAL , or nothing if the object does have the attribute PLURAL .
<i>object_label{sub}</i>	Prints the word you if the object is the current player, he if the object has the attribute ANIMATE , she if the object has both the ANIMATE and the FEMALE attribute, it if the object has neither the ANIMATE nor the FEMALE attribute or the word they if the object being referred to has the attribute PLURAL .
<i>object_label{obj}</i>	Prints the word yourself if the object is the current player, him if the object has the attribute ANIMATE , her if the object has both the ANIMATE and the FEMALE attribute, it if the object has neither the ANIMATE nor the FEMALE attribute or the word them if the object being referred to has the attribute PLURAL .

Custom Macros

It is also possible to define your own custom macros by creating a function with the name **+macro_** followed by the name of the macro. This is primarily of use when writing a JACL game or library using a language other than English. The object the macro is being applied to will be passed to the function as an argument. The output of the macro is specified by setting the value of the string variable **return_value**. For example:

```
{some_function
write noun1{The} " " noun1{is} " verrouillÃ©" noun1{es} ".^"
}

{+macro_es
if arg[0] has FEMALE
  if arg[0] has PLURAL
```

```
        setstring return_value "es"
    else
        setstring return_value "e"
    endif
else
    if arg[0] has PLURAL
        setstring return_value "s"
    else
        setstring return_value ""
    endif
endif
}
```

As you can see, the macro is used using the name **{es}** while the function is defined using the full name of **+macro_es**. The text to be returned by the macro is placed in the string variable **return_value** which is created internally by the interpreter.

Printing the Value of Strings

If the value of a string is to be printed, as opposed to verbatim text, the name of the string variable or string constant must be entered as a separate parameter that is not enclosed in quotes. For example:

```
constant game_title "Tutorial Game"
constant game_author "I. F. Author"

{+intro
write "Welcome to " game_title " by " game_author ".^"
}
```

The output of the **+intro** function will be:

```
Welcome to Tutorial Game by I. F. Author.
```

The PRINT Command

The **print** command is the primary way to output plain text. Unlike the **write** command, the values of variables, macros or string constants cannot be displayed. In fact the only thing that can be displayed by a **print** command is plain literal text. A **print** command has no parameters. On executing a **print** command the interpreter will begin printing all the text from the following line onwards until it encounters a line with a period (.) as the first non-whitespace character. The lines of text will be displayed verbatim with the following considerations:

1. if a line does not end with a caret (^), an implicit space will be printed after the line and the following line of text will continue straight after it, following the usual word wrapping rules;
2. if a line ends with a backslash (\) or a caret (^), no implicit space will be added to the line;
3. a vertical bar will be translated into a space to allow formatting such as paragraph indenting to be done.

Below is an example of the **print** command in action (note that the colon after the word **print** is optional whitespace):

```
{+intro
clear
print:
  This text will be printed to the screen and
  word wrapped in the appropriate places.
  The backslash at the end of this line will prev\
  ent an implicit space being printed as will
  the caret at the end of this line.^
  ||These two vertical bars will allow this line
  to be indented two spaces.^
.
}
```

The LOOK Command

The **look** command will print the long description of the current location and any visible objects that are in the current location. A **look** command is executed implicitly by the JACL interpreter whenever a description of the player's current location is required. This is at times such as when the player moves into a location for the first time or restores a saved game.



If the player has set the game to verbose mode (**DISPLAY_MODE** = 1), locations will not be given the attribute **VISITED**.

The very first thing the **look** command will do is remove the attribute **VISITED** from the current location. It will then execute the global function **+before_look**. If this function exists, and returns **true**, nothing more is done. If it does not exist, or returns **false**, the function **+title** will be executed. This function is an opportunity to print extra information that is potentially relevant to all locations. For example, the **+title** function in **frame.jacl** tests if the player is currently sitting down and displays the text "(sitting)". It is also common to add the following line to the top of **+title** to give each location description a title:

```
write here{The} ^
```

Next the **look** function associated with the current location is executed. After this has finished, the interpreter will display the text supplied by the **long** property for each object that is in the current location and doesn't have a **mass** of **scenery**.

If the **long** property of an object is set to **function**, that function will be executed. This is useful for descriptions that are either too long to fit in a single line of code or change during the course of the game. If the **long** property is not set to **function**, the property text is displayed verbatim.

The final step in the process is that the global function **+after_look** is executed.

It is a convention to have a **look** command at the end of the **+intro** function so the player can see where they are and what objects are nearby when the game begins.

The MORE Command

The **more** command will print a message and then wait for the player to press a key before continuing. The message to print is passed as the only parameter of a **more** command. If no message is provided, the message **[MORE]** is used as a default. Below is an example of using the **more** command with a custom message:

```
more "Hit any key to continue"
```


Glk and Multimedia

Glk is a portable application programming interface (API) for applications, like JACL, with a predominantly text-based user interface. By communicating with this API, JACL is able to work with a variety of user interfaces implemented by third parties for a variety of platforms. The Glk specification was written by Andrew Plotkin, and many thanks to him for having done so. Of equal importance are the various implementations of the Glk specification.

So what does all this mean? The Glk specification defines a set of multimedia functionality that the native console version of JACL does not provide. Currently JACL can be compiled using WindowsGlk by David Kinder and Gargoyle by Tor Andersson. Many thanks to both these gentlemen for writing their libraries and assisting in the porting of JACL to use them. The commands detailed below are available for use by a JACL interpreter that is compiled with either WindowsGlk or Gargoyle. In the standard Unix distribution, the JACL interpreter is also compiled with GlkTerm. GlkTerm is a Glk library written by Andrew Plotkin using ncurses. GlkTerm does not support graphics or sound. The multimedia commands detailed in this chapter will be ignored without error by interpreters that do not support multimedia.

If you would like to have your game behave differently is certain Glk features are supported by the interpreter, you can test whether the integer constants **graphics_supported**, **sound_supported** and **timer_supported** are set to **true** or **false**.

Graphics, sounds and timers can also be turned off from within your game by setting the integer variables **graphics_enabled**, **sound_enabled** and **timer_enabled** to **false**.

Some of the text for this chapter has been cheerfully stolen from the Glk specification. The full Glk Specification can be read at <http://www.eblong.com/zarf/glk/>.

Blorb Files and the bjob Utility

All sound and image files are made available to a Glk interpreter by placing them in a Blorb file with the same base name as the game. For example, the game **grail.jacl** will look in the file **grail.blorb** for any sounds or images required.

Blorb is a second specification written by Andrew Plotkin specification for a common format for storing resources associated with interactive fiction games. There are many tools available for creating Blorb files. A program called **bjob** is included in the JACL package that is a slightly modified version of the utility **blc**. **blc** is part of the iBlorb suite developed by Ross Raszewski. Many thanks to Ross for both his original development effort and the permission to include this program in the JACL package. Information about iBlorb can be found at <http://www.trenchcoatsoft.com/projects.html>. The full Blorb Specification can be read at <http://www.eblong.com/zarf/blorb/>.

The utility **bjob** creates a blorb file with the help of a **.blc** control file. This control file is a plain text file that specifies the sounds and images to include in the blorb file. Each line in a **.blc** control file describes one chunk of the Blorb file, and has the following format:

Use IndexNumber Type File

Use is the usage of the resource and can either be **Pict** or **Snd**.

IndexNumber is the number you will use to refer to this resource in your program.

The JACL Author's Guide

Type is the resource type. This can be either JPEG, PNG, FORM (Aiff), OGGV or MOD.

File is the name of the file to be included as this resource.

Below is an example **.blc** file called **example.blc**:

```
Pict 1 PNG /images/title.png
Snd 3 MOD /music/theme.mod
Snd 4 OGGV /sounds/explosion.ogg
Pict 2 JPEG /images/car.jpg
```

To create a Blorb file from this **.blc** file, use the **bjorb** utility in the following manner:

```
bjorb example.blc example.blorb
```

This command will create the Blorb file **example.blorb** that will be automatically read by the game **example.jacl**. When the **bjorb** utility runs, it will output some JACL code that will create a constant for each image or sound. If you find these convenient you can cut and paste this code into your game.

For example, when reading this **.blc** file:

```
Pict 1 PNG images/blackjack.png
Pict 2 PNG images/chip25.png
Pict 3 PNG images/chip50.png
Pict 4 PNG images/chip100.png
Pict 5 PNG images/club.png
Pict 6 PNG images/diamond.png
Pict 7 PNG images/spade.png
Pict 8 PNG images/heart.png
```

bjorb will produce the following output:

```
bjorb 1.0 (Apr 30 2008) by Stuart Allen, based on
Blorb Packager Version .5b by L. Ross Raszewski
```

```
# CONSTANTS FOR RESOURCES IN BLORB FILE
constant IMAGE_blackjack 1
constant IMAGE_chip25 2
constant IMAGE_chip50 3
constant IMAGE_chip100 4
constant IMAGE_club 5
constant IMAGE_diamond 6
constant IMAGE_spade 7
constant IMAGE_heart 8
```

If you pass only a single command-line argument to **bjorb**, that argument will be used as the base name for both the **.blc** control file and the **.blorb** output file. As an example, the following two commands are equivalent:

```
bjorb example
```

```
bjorb example.blc example.blorb
```

The IMAGE Command

The **image** command is used to display one of the images stored in the game's blorb file. The image to display is specified using its index in the blorb file. The index can be supplied either as a literal integer or any JACL container that resolves to an integer. The index of the image can optionally be followed by an alignment. The possible alignments are:

Alignment	Description
up	The image appears at the current point in the text, sticking up. That is, the bottom edge of the image is aligned with the baseline of the line of text.
down	The image appears at the current point, and the top edge is aligned with the top of the line of text.
centre	The image appears at the current point, and it is centered between the top and baseline of the line of text. If the image is taller than the line of text, it will stick up and down equally.
left	The image appears in the left margin. Subsequent text will be displayed to the right of the image, and will flow around it -- that is, it will be left-indented for as many lines as it takes to pass the image.
right	The image appears in the right margin, and subsequent text will flow around it on the left.

The two "margin" alignments require some care. To allow proper positioning, images using **left** and **right** must be placed at the beginning of a line. That is, you may only call **image** (with these two alignments) if you have just printed a newline, or if the screen is entirely empty. If you margin-align an image in a line where text has already be printed, no image will appear at all.

The following code demonstrates the **image** command:

```
constant IMAGE_house 4

{+display_image
# DISPLAY IMAGE 4 WITH THE TOP OF IMAGE LEVEL WITH TOP OF THE TEXT
image IMAGE_house

# DISPLAY IMAGE 6 WITH THE BOTTOM OF IMAGE LEVEL WITH TOP OF THE TEXT
image 6 up
}
```

The SOUND Command

The **sound** command is used to play one of the sounds stored in the game's blorb file. The sound to play is specified using its index in the blorb file. The index can be supplied either as a literal integer or any JACL container that resolves to an integer.

The index of the sound can optionally be followed by a channel to play the sound on. There are eight available channels numbered zero to eight (0 -8). If no channel is specified, channel 0 is used as the default.

If a channel is specified, it can be followed by the number of times to repeat the sound. If **-1** is specified as the number of times to repeat the sound, the sound will keep playing until it is manually stopped using the **stop** command. If no number of times to repeat the sound is specified it is played once.

The following code demonstrates the **sound** command:

```
constant SOUND_rain      6
constant SOUND_thunder  7
integer  AUDIO_CHANNEL 2

{+play_sound
# PLAY SOUND 7 ON CHANNEL 0 ONCE ONLY
sound SOUND_thunder

# PLAY SOUND 6 ON CHANNEL 2 ONCE ONLY
sound SOUND_rain AUDIO_CHANNEL

# PLAY SOUND 6 ON CHANNEL 3 FOUR TIMES
sound 6 3 4

# PLAY SOUND 6 ON CHANNEL 3 INDEFINITELY
sound SOUND_rain 3 -1
}
```

The VOLUME Command

The **volume** command is used to set the volume of a sound channel. The volume is specified as an integer between 0 and 100. A second, optional parameter can be used with the **volume** command specifying which sound channel to set the volume for. If this parameter is omitted the volume is set for channel 0.

The following code demonstrates the **volume** command:

```
{+set_volume
# SET CHANNEL 0 TO FULL VOLUME
volume 100

# SET CHANNEL 2 TO HALF VOLUME
volume 50 2
}
```

The STOP Command

The **stop** command simply stops the sound being played on the specified sound channel. The channel to stop is specified as an integer between 0 and 3. If no channel is specified channel 0 is stopped by default.

The following code demonstrates the **stop** command:

```
{+stop_sound
# STOP THE SOUND PLAYING ON CHANNEL 0
stop

# STOP THE SOUND PLAYING ON CHANNEL 2
stop 2
}
```

The TIMER Command

The **timer** command tells the interpreter to call the function **+timer** every so many milliseconds, regardless of whether the player types a command or not. The number of milliseconds to wait between each function call is

specified as the **timer** command's only parameter. If you specify a time of **0** to a **timer** command, the timer will be turned off.



It is important to be aware that not all interpreters will support the **timer** command, so no processing essential to the game should be performed within the **+timer** function. Periodically playing sound effects that are not essential to the game is an example of valid use of this functionality.

Below is an example of the **timer** command being used:

```
{+intro
...
# SET THE TIMER TO EVERY TEN SECONDS
timer 10000
...
}

{+timer
if here has OUTDOORS
  # PLAY THE THUNDER SOUND ON CHANNEL 3 SO
  # IT DOESN'T INTERFERE WITH OTHER SOUNDS
  play SOUND_thunder 3
endif
}
```

The STYLE Command

The **style** command is used to output either ANSI terminal codes or set Glk styles depending on the interpreter being used. The **style** command accepts a single string containing the name of the style to set. Here is an example of using the **style** command to output some bold text:

```
style bold
write "This is bold.^^"
style normal
```

Below is a table showing styles available in JACL and the Glk styles they map to:

JACL Style	Glk Style
bold	style_Emphasized
note	style_Note
input	style_Input
header	style_Header
subheader	style_Subheader
reverse	style_Note
pre	style_Preformatted
normal	style_Normal

The Status Window

The status window at the top of the screen is implemented as a Glk window, and is therefore covered here. It is possible to have no status window at all, use the built-in, default status window or design a custom one

The JACL Author's Guide

yourself. The vertical height of the status window in rows is defined by using the constant **status_window**. If the constant **status_window** is set to **0**, no status window will be created. For example, to create a status window that is three rows tall, define the following constant:

```
constant status_window 3
```

The status window is always created using a fixed-width font and its current dimensions can be read at any time using the values of **status_width** and **status_height**. Although **status_height** will most often be equal to **status_window** (unless **status_window** exceeded the available space), **status_width** will change as the player's resizes the game window.

If a status window is created, the interpreter will attempt to call the global function **+update_status_window**. If this function exists, it will be executed and must contain code to draw the contents of the status window. If this function doesn't exist, the interpreter will use internal code to generate a standard interactive fiction status line. This consists of a single line with the name of the current location against the left side and the number of moves and current score against the right.

If the function **+update_status_window** does exist, the current Glk stream is set to that window before it is called so all **write** or **print** commands will output to it. The window is also first cleared of all previous contents and the cursor is positioned in the top left corner.

The location at which to start printing text within the status window is changed using the **cursor** command. The **cursor** command is passed two integer parameters, the row and column to move the cursor to. Counting starts in the top left corner at 0, 0. For example, to move the cursor to the far right hand column of the second row, use the command:

```
cursor status_width 1
```

Often when positioning text it is important to know the length of the string you are going to print. This is determined using the **length** command. The **length** command requires two parameters: the container to hold the length of the string and the string itself. For example, the following command determines the length of the string constant **game_title** and stores the result in the variable **index**:

```
length index game_title
```

Putting this all together, it is possible to display the title of the game centred in a single-line status window using the following code:

```
constant game_title "The Unholy Grail"
```

```
constant status_window 1
```

```
integer index  
integer offset
```

```
{+update_status_window  
set offset = status_width  
length index game_title  
set offset - index  
set offset / 2  
cursor offset 0  
write game_title  
}
```


The JACL Author's Guide

Status windows are often displayed using reverse text to make it clearly stand out from the main window. This is achieved using the following command:

```
style reverse
```

This command, however, will only reverse the text output, not the whole window. In order to achieve the effect of an entirely reversed window, blank spaces will need to be printed wherever there isn't any other text. The easiest way to do this is to print entire rows of blank spaces then move the cursor back to print over the top. The **padstring** command exists to help with this process. The **padstring** command takes three parameters. The label of the string to fill, the text to fill the string with and an integer specifying the number of times to copy the text into the string. To print a blank line in the status window, the text that will be copied is a single space in quotes (" ") and it will be copied **status_width** times. The code below is an expanded version of the above function that prints the title of the game centred in an inverse status window:

```
constant game_title "The Lovely Test Game"
string status_text

integer index
integer offset

constant status_window 1

{+update_status_window
style reverse
padstring status_text " " status_width
write status_text
set offset = status_width
length index game_title
set offset - index
set offset / 2
cursor offset 0
write game_title
}
```

As a final example of a **+update_status_window** function, below is the JACL code to replicate the internal status line produced if no custom function is provided:

```
string status_text

integer index

constant status_window 1

{+update_status_window
style reverse
padstring status_text " " status_width
write status_text
cursor 1 0
write here{The}
setstring status_text "Score: " score " Moves: " total_moves
set offset = status_width
length index status_text
set offset - index
set offset - 1
cursor offset 0
write status_text
}
```

The UPDATESTATUS Command

The interpreter will call the **+update_status_window** function after each of the player's moves and when the game window is resized. If you require the status window to be updated at other times such as in a loop or from the **+timer** function, use the **updatestatus** command. The **updatestatus** command takes no parameters. It sets the current output stream to the status window and clears the status window before calling the function **+update_status_window**. When **+update_status_window** has finished executing the current output stream is set back to the main window.



It is not possible to call the **+update_status_window** directly. It will be called by the interpreter automatically after each of the player's moves or when the window is resized. If you do require the window to be updated at other times use the **updatestatus** command. If you call **+update_status_window** directly the current output stream will not be set to the correct window.

HTTP and HTML

This chapter covers the aspects of HTML and HTTP that relate to writing games for use with the **cgijacl** interpreters, including the JACL commands that assist with the output of HTML. If you are not familiar with HTML, it is highly recommended that you read at least a basic tutorial before starting to write your first web-based game.



The **jacl** interpreter will consider all the HTML-specific commands as errors so it is important to either test the value of constant **interpreter** (either **GLK** or **CGI**) to be sure the game is being played with a web version of JACL or use these commands from a function like **+header** or **+footer** that is only executed by the **cgijacl** interpreter.

Document Structure

A complete HTML page must be returned by your game in response to each of the player's moves. To assist with this, if you do not define your own custom header and footer functions (**+header** and **+footer**), the **cgijacl** interpreter will insert template headers and footers to mimick a standard interactive fiction interface. If a **+header** function is supplied, it is the very first thing to be executed. The default header looks like this:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">

<html><head>
  <title>game_title</title>
  <script language="JavaScript">
    <!--
    function userCommand() {
      var xhReq = createXMLHttpRequest();
      if(xhReq == null) { return true; }
      var user_id = document.JACLGameForm.user_id.value;
      var command = document.JACLGameForm.command.value;
      xhReq.open("GET", "?user_id="+user_id+"&command="+command+"&ajax=true", false);
      xhReq.send(null);
      var serverResponse = xhReq.responseText;
      var maintext = document.getElementById("maintext");
      maintext.innerHTML += "<br><b>" + command + "</b><br>" + serverResponse;
      maintext.scrollTop = maintext.scrollHeight;
      document.JACLGameForm.command.value = "";
      return false; }
    function putFocus(formInst, elementInst) {
      if (document.forms.length > 0) {
        document.forms[formInst].elements[elementInst].focus(); }}
    function createXMLHttpRequest() {
      try { return new XMLHttpRequest(); } catch(e) {}
      try { return new ActiveXObject("Msxml2.XMLHTTP"); } catch (e) {}
      return null; }
    -->
  </script>
  <style> <!--
    #footer {
      position:absolute;
      bottom:0;
      left:0;
      right:0;
      display:block;
      font-family: Verdana, Arial, Sanserif;
      padding-top: 10px;
```

The JACL Author's Guide

```
padding-bottom: 10px;
padding-left: 20px;
background: #dddddd; }
#main {
position: absolute;
margin-left: 10px;
margin-right: 10px;
left: 0;
top: 0px;
right: 0;
overflow: auto; }
div.maintext {
font-family: Verdana, Arial, Sanserif;
padding-top: 20px;
padding-bottom: 20px;
padding-left: 50px;
padding-right: 50px;
font-size: 12px;
overflow: auto; }
--> </style>
</head><body onLoad="putFocus(0, 0);">
<div id="main">
<div id="maintext" class="maintext">
```

This header sets the DOCTYPE for the returned HTML document, sets the title of the browser window to the name of the game, provides some Javascript functions to set the input focus to the command prompt and make the Ajax call to the server when a command is entered. It also outputs the CSS styles required and opens a DIV for the game's output to appear in.

The body of the **maintext** DIV is then produced by your game.



If you do provide your own custom **+header** function, be aware that you can't display values that are modified by the player's command. This is because a full page is generated in order, top to bottom. For example, if the header was to display the title of the current location, when the player moved the header would be displayed and would show the current location. The movement command would then be processed to move the player and then the footer displayed. As you can see, this would result in the header showing the location the player was in *before* the command took place. The footer does not have this limitation.

The function **+footer** is the very last function to be executed after a player's move has been processed. If one is not supplied the following default output is inserted by the interpreter:

```
</div>
<div id="footer" class="footer">
  <p><b>></b> <input type="text" style='border-style:none;' size=60 name="command"></center>
  <input type="hidden" name="user_id" value="xxxxxxx-xxx">
</div>
</form>
</body>
</html>
```

There are two significant aspects to this footer. The first is that it contains the input field where the player will type their commands. This input field must have the name **command** as this is where the interpreter will look for the player's command when the form is submitted. The second is the hidden input that contains the **user_id** for the player.



An alternative to using the header and footer built into interpreter or coding your own from scratch is to use the [webinterface.library](#).

Linebreaks

In order to avoid the need to fill a standard interactive fiction game or the standard library with HTML tags the caret newline character (^) is output by the **cgijacl** interpreters as **
** followed by a newline. This behaviour can be turned off by setting the JACL variable **linebreaks** to **false**. This is usually desirable in sections such as the header and footer where code is only executed by the web-based interpreters and can therefore contain any amount of custom HTML you require.

The Player's User ID

HTTP is a stateless protocol. This means that each request received by the web server is completely independent of any that came before it. The **cgijacl** interpreters, however, need to know which player each request has come from, and more specifically, what state the game world was in at the end of their previous move. For this reason, a unique user ID must be sent with each of the player's commands. Failure to do so at any time will result in the game restarting from the beginning and a new user ID being assigned automatically.

To propagate the player's user ID, it must be passed as the HTML parameter **user_id**. This is done by including it as a hidden field in any forms, or as a URL parameter in any hyperlinks. The token **\$user_id** maybe be used in any **write** statement to output the automatically generated ID. Below are two examples of its use:

```
write "<a href=~" $url "?user_id=" $user_id
write "&command=look~>Look</a>"

write "<input type=~hidden~ name=~user_id~ "
write "value=~" $user_id "~>"
```

In the first set of **write** statements, the token **\$user_id** is added directly to the end of the URL being linked to as the value of the HTTP parameter **user_id**. Also used is the token **\$url**. This token displays the URL of the game, such as **/fastcgi-bin/game.jacl**.



When playing games with the **cgijacl** interpreter, there is no URL to the game. It is wise, however, to always construct links using the **\$url** token so your game will also work when played with the **fcgijacl** interpreter.

The second set of **write** statements output a hidden form field. This field should be used as part of an HTML form when the player issues their commands by submitting this form.

When using an HTML form, there are two simpler methods of including the player's user ID. The first is the command **hidden**. This command outputs a line exactly like the **input** tag above. Below is an example of its use:

```
write "<form>"
write "<input type=~text~ name=~command~>"
hidden ;adds the player's user ID as a hidden form field.
write "</form>"
```

As you can see, there is nothing difficult about using the **hidden** command. The second related command is **prompt**. This command outputs a text input box and the player's user ID. Using **prompt**, the above code snippet would look like this:

```
write "<form>"
prompt ;adds a text field with the name 'command' and
      ;the player's user ID as a hidden field.
write "</form>"
```

As you can see, there is even less to using the **prompt** command.



If you are running a game using the **fcgijacl** interpreter under a full webserver, the value of the environment variable **REMOTE_USER** will be used if present instead of creating a randomly generated user ID. This allows a JACL game to be protected by the webserver and the player to receive the same user ID each time they log in and play.

The Player's Commands

The player's commands are passed to the interpreter via three parameters: **verb**, **noun** and **command**. If the parameters **verb** and **noun** are passed, **noun** is concatenated on the end of **verb** and the result is processed as a single command. This is useful when writing point-and-click adventures that use a combination of list boxes and submit buttons to form verb-noun commands. If the parameter **command** is passed, it is processed as a complete command. This is useful for traditional text-based adventures.

It is also possible to use both methods of passing the player's moves. When a request is received, the interpreter will first test the value of **verb**. If this is empty, the value of **command** is used as the player's move. If **verb** does contain a value, the value of **noun** will be concatenated to it, and the move will be processed, ignoring any value that **command** may or may not have.

Ajax Requests

Using standard Javascript techniques (see the default header or **webinterface.library** for an example), it is possible to request the JACL interpreter to process a command without refreshing the whole browser window. Once the command has been processed Javascript must be used once again to manually insert any output returned into the appropriate HTML element. When performing these types of requests, there are two special URL parameters to be aware of.

The first is the **ajax** parameter. If this parameter has the value **true** the header and footer functions will not be called, leaving the interpreter to return only the output produced directly by the command being processed.

The second is the **function** parameter. This parameter allows the game author to specify a function to execute as opposed to the usual process of interpreting a command entered by the player. The value of the **function** parameter is the name of the function to execute. For security purposes, the acceptable values for this parameter are restricted to a set number of functions: **+timer**, **+ajax** and **eachturn**. The first two options simply call the function of the same name and return any output. The third will call both the global function **+eachturn** and the **eachturn** function that is associated with the current location.



The **+ajax** function serves no predetermined purpose, it is simply a function that can be called by the game author to handle any special-case situations that arise.

The BUTTON Command

The **button** command is used to create a button that will submit an HTML form. It must be followed by a single parameter that indicates what command this button should issue. Behind the scenes this command creates an **input** tag of the type **submit** and sets the value of the HTML parameter **verb** to the specified command. Below is an example of its use:

```
button Look
button "Read Map"
```

The command specified for each button will appear as text on the button. If a command contains any spaces, the command must be enclosed in double quotes as usual.

It is possible to use a second HTML control, such as a list box, in conjunction with a submit button to create verb-noun commands. See the **control** command below for more information.

The HYPERLINK Command

The command **hyperlink** is a simple method of creating a hyperlink that issues a command for the player. It can accept either two or three parameters. The first is the visible text to be displayed. The second parameter is the command to be issued when the link is clicked. The third, optional, parameter is the name of a CSS class that should be applied to the link. Below are some sample hyperlinks:

```
hyperlink East east
hyperlink "Take All" "take+all" small
hyperlink "Where am I?" look big
```

With the third hyperlink, the text **Where am I?** will appear on the screen with all the styles specified the CSS class **big** applied. When clicked, it will pass the value **look** to a parameter named **command**. This, in effect, issues a **look** command on the player's behalf.



When using the **hyperlink** command, any spaces in the third parameter (the command to be executed), must be replaced with plus signs. This is due to the fact that the command is pasted as part of the URL sent to the web browser, and spaces are not valid in a URL.

When manually creating special-purpose hyperlinks, the **{+name}** macro is very useful. For more information, see the chapter on [Screen Display](#).

The CONTROL Command

The **control** command is also used to create a hyperlink. With the **control** command, however, the first parameter is the name of an image to display instead of some plain text. As the **control** command displays an image, no third parameter indicating a CSS style is allowed. Below is an example of a **control** command:

```
control "/images/look.png" look
```

This command will create a hyperlink that issues the command **look**. The hyperlink will appear on the screen as the image **look.png**.

The OPTION Command

The **option** command is used to include an object as a part of a **<select>** list box. A list box allows the user to select one of the objects from the list then submit the form. The selected object is then sent to the server in the form of a name and value pair. This is most useful when the **select** tag used to create the list box sets the object selected to the parameter **noun** as the **cgijacl** interpreters will automatically look for this parameter when constructing a move to process. By default, the **option** command will set the value returned to all of the specified object's names. The object for each option is specified by following the **option** command with an object label or pointer. The visible text for the option is the object's short description. For example:

```
<form>
  write "<select name=~noun~>"
  loop
    if noun3 is *present
      option noun3
    endif
  endloop
  write "</select>"

  button Look
  button Use
  button Take
  button Drop
</form>
```

The above code creates a list box with a parameter name of **noun** and adds an entry for each object that is in the player's current location. If an object is selected before the form is submitted by clicking one of the four buttons, the parameter **noun** will have its value set to all the names of the object the player selected and the parameter **verb** will have its value set to the text on the button that was clicked. These two parameter would then be concatenated (verb + noun) and processed by the interpreter as the player's move.

It is also possible to specify that an **option** command should pass the object's index as opposed to its names. This is done by following the object pointer or label with the word **index**. This is only of use if a **parameter** statement has been defined to accept the parameter name specified in the **select** tag. Below is an example of this:

```
parameter destination SOME_VARIABLE

<form>
  write "<select name=~destination~>"
  loop
    if noun3 is *present
      option noun3 index
    endif
  endloop
  write "</select>"
  hidden
  button Jump
</form>
```

When this form is submitted, the **cgijacl** interpreter will look for the parameter **destination**, as it has been defined using a **parameter** statement. This parameter will contain the internal number of the object that was

selected and this number will be copied into the location specified by the **parameter** statement. In this case, a variable called **SOME_VARIABLE**. For more information see the section on [parameters](#).

The IMAGE Command

The **image** command is used to display an image — no surprise there. It requires three parameters using the following syntax:

```
image URL Alignment AlternateText
```

For example:

```
image "/images/view.jpg" left "[View from the balcony]"
```

This command displays the image **/images/view.jpg** aligned to the left. If the picture is not found, or is displayed in a text-only browser such as Lynx, the text **[View from the balcony]** will be displayed instead. This is also the text that will be displayed by most graphical browsers when the cursor is held over the image.

The Media File

When played with the **cgijacl** interpreter, each game will require a corresponding **.media** file in order to serve multi-media content such as images. This file must have the same name as the game, with a **.media** suffix in place of the **.jacl** suffix. For example, the game **grail.jacl** has a matching media file called **grail.media**. As this game only has one image, its media file only has one line:

```
/grail.jpg image/jpeg images/grail.jpg
```

Each line of a media file has three columns separated by white space. The first column is the URL of the file as specified in the HTML of your game. The second column is the file's MIME type. The third column is the name of file on the local system to send when this file is requested. Files not beginning with a forward slash are relative to the directory containing the gamefile.

If **awk** is installed on your system, the media file can be automatically generated from the **.jacl** game file using the **build-media-file** script in the **bin** directory of the JACL distribution. This script accepts the name of the game file as its only parameter and sends its output to standard out. It also assumes that all files live in an **images** directory beneath the directory containing the game file. **build-media-file** is used with a command like:

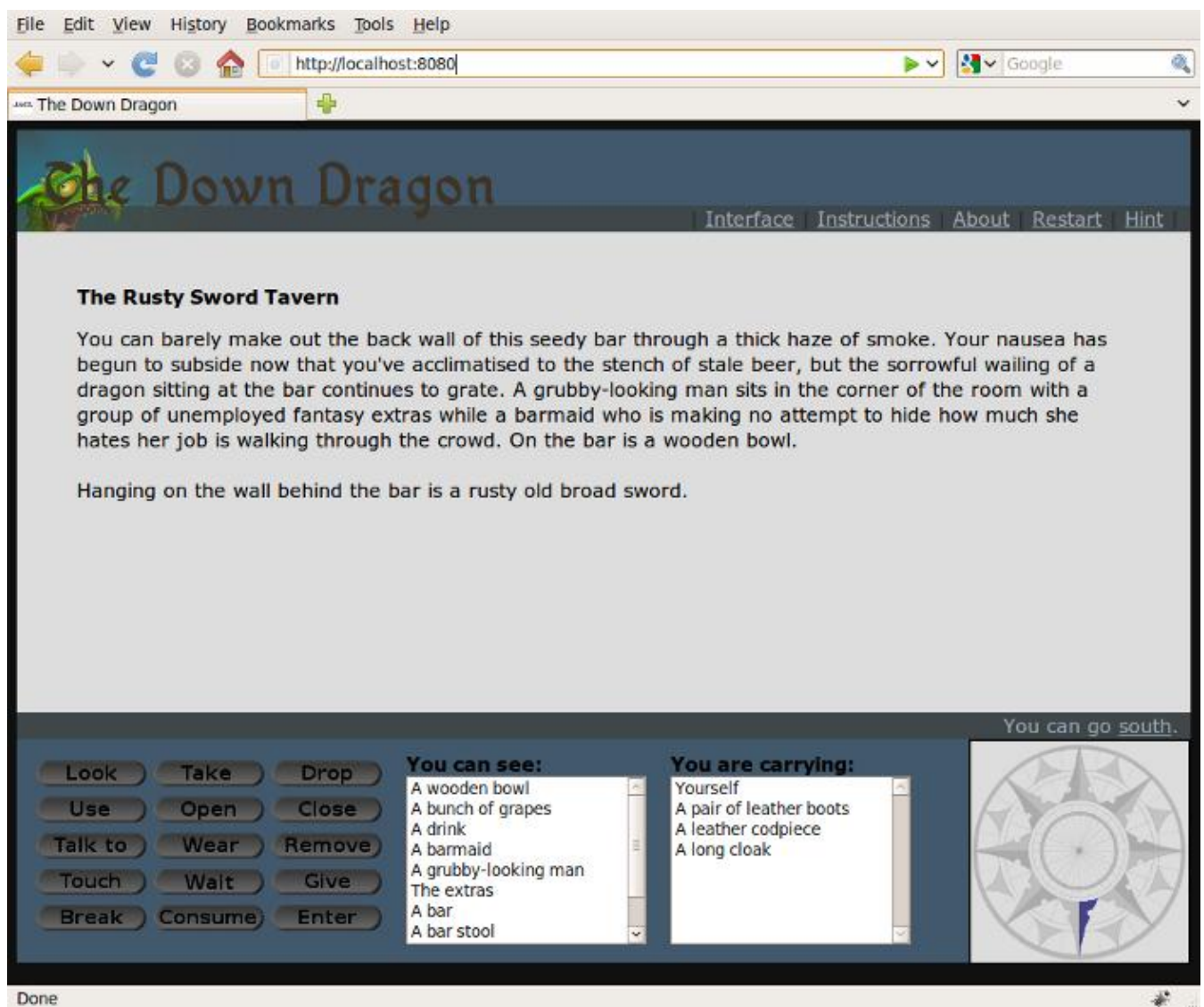
```
build-media-file grail.jacl > grail.media
```

JACL games played using the **fcgijacl** interpreter and a full webserver do not require a **.media** file. The path to the image specified in the HTML of your game will be looked for in the document root of your webserver.

GUI Web Interface

A set of library files are included with JACL to assist in the creation of a more visually pleasing and flexible web interface for your games. These libraries display an interface that operates in one of three modes. The first is the Ajax mode which is functionally equivalent to the default header and footer with the addition of control over colours and images and hyperlinked exits. The second is the Standard mode which does not use Ajax for the command submission and includes a compass rose in the footer. The third is the GUI mode which displays two list boxes, one for the objects being carried and one for the objects in the current location; a compass rose for moving around; and a set of action buttons that represent some of the common verbs. When this interface is used, a text command is constructed and processed by the interpreter as normal. The player is given the choice of swapping between these three modes.

Below is a screen shot of the web interface running in GUI mode:





If you wish your game to be playable using this point-and-click interface you must be careful to design it to be solveable using only the verbs available.

Using the Interface

To use this web interface you first need to include the appropriate library at the bottom of your game file with the line:

```
#include "webinterface.library"
```

Once this library is included, certain string constants must be defined to provide information that is specific to your game. Here are the values from The Down Dragon, as show in the screenshot above:

```
constant title_image "/images/dragon.png"  
constant footer_image "none"  
constant header_colour "#42596d"  
constant linkbar_colour "#3f3527"  
constant maintext_colour "#ddddd"
```

The value for **title_image** specifies which image to display in the top title bar. This image must be 85 pixels high and may be of any width. As browsers can be resized by the player, best results are achieved by having the right-hand edge of the image blend well with the background colour specified in **header_colour**. Note that this is also the colour used for the footer of the page. If **image_title** is set to **none**, the value of **game_title** is displayed instead.

Like the header it is also possible to choose an image to be displayed in the footer of the page. This image is displayed in the background behind the user interface itself. This image is set to repeat both horizontally and vertically.

The value for **main_text** configures the background colour of the main window that contains the game's text output.

Colours can be specified in any format acceptable to the HTML specification. For example, it is also valid to supply a plain text colour such as **blue**. The colours above are specified as hexadecimal RGB values to provided more fine-grained control over the exact shade of the colour.

Flow Control

The IF, IFALL and ENDIF Commands

An **if** command tests whether an expression is true or false for the purpose of selectively executing code. If the expression is true, execution will continue from the next line. If the expression is false, execution will continue from the line after the matching **endif** or **else** command. If no **endif** command is found by the end of the function, the function will terminate normally with an implicit **return true**. An **if** command must be of the format:

```
if LeftValue test RightValue [LeftValue Test RightValue]...
```

In an **if** command, each set of three parameters (two values and a test), is called an expression. Each expression is evaluated to equal either true or false. If more than one expression is supplied, the entire statement is considered be true if any one of the expressions is true — a logical **OR**. The **ifall** command has the same format as an **if** command except the entire statement is only considered to be true if all of the expressions are true — a logical **AND**.

The following table lists the possible tests that can be used with an **if** or **ifall** command when the left and right values are both either an integer, an integer constant, an object or the word **random**. For more information on constants and **random** see the chapter on [Internals](#).



Each object defined is assigned a unique number when the game is loaded. In a **set**, **if** or **ifall** command, an object's label is substituted for this number. Objects are numbered in the order they appear in the game file starting at one.

Test	Description
= or ==	This tests if the left value is equal to the right value.
>	This tests if the left value is greater than the right value.
<	This tests if the left value is less than the right value.
>= or =>	This tests if the left value is equal to or greater than the right value.
<= or =<	This tests if the left value is equal to or less than the right value.
!= or <>	This tests if the left value is not equal to the right value.

The following are the possible tests that can be used with an **if** command when the left and right values are both set to an object:

Test	Description
grandof	This tests if the left object is the grand parent of the right object.
!grandof	This tests if the left object is not the grand parent of the right object.

When an object is placed inside another object, which is then placed inside yet another object, the last object is said to be the grand parent of the first. This is the case no matter how many intermediate objects there are.

The following are the possible tests that can be used with an **if** command when the left value is a location and the right value is an object:

Test	Description
locationof	This tests if the left value is the location of the right object.
!locationof	This tests if the left value is not the location of the right object.

The difference between **grandof** and **locationof** is that, for example, if a key is put on a keyring, which is put in a box, which is put into a bag, and the bag is in a room called "beach", '**locationof key**' would refer to the beach, but '**grandof key**' would refer to the bag.

The following are the possible tests that can be used with an **if** command when the left value is an object and the right value is an attribute. For more information see the chapter on [Attributes](#).

Test	Description
has	This tests if the object has the specified attribute set
hasnt	This tests if the object hasn't the specified attribute set

The following are the possible tests that can be used with an **if** command when the left value is an object and the right value is one of the words ***here**, ***held**, ***present** or ***anywhere**. These words have the same meanings when used in an **if** statement as in a **grammar** statement. For more information see the section on [Grammar Statements](#).

Test	Description
is	This tests if the object is in the specified scope.
isnt	This tests if the object isnt in the specified scope.

To help clarify, here are some examples of the various types of **if** commands:

```
if beach locationof bucket
if sand grandof bucket
if TOTAL_MOVES >= 42
if glove has WORN
if guard isnt *present : id_card has WORN
if noun1 = sword : noun1 = knife
if sword(parent) = field
```

It is possible to nest **if** statements. Nesting involves placing a second **if** command before the matching **endif** command of a first. The end result of this is that the code between the second **if** command and its matching **endif** command will only be executed if both statements are true — a logical **AND**.

The IFSTRING Command

An **ifstring** command is used to compare two strings of text. If the string of text is to contain any spaces, it must be enclosed in double quotes. An **ifstring** command must use the following format:

```
ifstring text test String [text Test String]...
```

Below is a list of the possible tests:

Test	Description
== or =	This tests if the first string equals the second string

!= or <>	This tests if the first string doesn't equal the second string
contains	This tests if the first string contains the second string
!contains	This tests if the first string doesn't contain the second string
==C or =C	This tests if the first string equals the second string including case
!=C or <>C	This tests if the first string doesn't equal the second string including case
containsC	This tests if the first string contains the second string including case
!containsC	This tests if the first string doesn't contain the second string including case

Below are some examples of the **ifstring** command being used.

```
ifstring string_arg[0] contains "help"
ifstring command[0] == "take"
```

The IFEXECUTE Command

The **ifexecute** command works in a similar manner to the **call** command (see the chapter on [Functions](#) of more information.) If the function specified after the **ifexecute** command exists and does not return **false**, executing will continue from the line after the **ifexecute** command. If the function being called does not exist or returns **false** executing will continue from after the matching **endif** or **else** command.

This command is normally only used by library code to test if the game author has provided some specific associated function and to perform some sort of default action if not. For example:

```
ifexecute "here.look"
    # ASSOCIATED FUNCTION PROVIDED, DO NOTHING
else
    # NO ASSOCIATED FUNCTION, PERFORM DEFAULT
    execute "+look"
endif
```

If the **ifexecute** command is true, in other words, a **look** function associated with the player's current location exists and does not return **false**, the block of code below it will be executed. As the associated **look** function has already performed any required action, this block contains only a comment. If this function did not exist or returned **false** the function **+look** would be executed.

The ELSE Command

The code following an **else** command is executed only if the matching **if** command was false. If the matching **if** command was true, execution will continue from the line after the matching **endif** command. For example:

```
if mulder has DEAD
    write "Clamminess on your lips.^^"
else
```

```
    write "He looks quite surprised to say the least.^"  
endif
```

Nested **if** commands may also make use of the **else** command. This is demonstrated in the following section of code taken from the **+take_all** function:

```
loop  
  if noun3 childof noun1  
    if noun3(mass) < heavy  
      if noun3 hasnt LIQUID  
        if noun3(mass) <= player(info)  
          if TURN_WORKED = true  
            set TOTAL_MOVES + 1  
            execute "+eachturn"  
          endif  
          execute "+take_routine"  
          set INDEX + 1  
        else  
          write "You are carrying too much to take "  
          write noun3{the} .^  
          set INDEX + 1  
        endif  
      else  
        write noun3{The} " run" noun3{s} " through "  
        write "your fingers.^"  
        set INDEX + 1  
      endif  
    endif  
  endif  
endloop
```

In the above code, the block of code after the first **else** command is executed if the matching **if** command above it:

```
if noun3(mass) <= player(info)
```

is false. This line of code will not get executed, however, if the line:

```
if noun3 hasnt LIQUID
```

is false as execution will have already jumped to after the second **else** command.

The LOOP and ENDLOOP Commands

The **loop** command is used to iterate through all the objects (and locations) defined in the game. The **loop** command takes a single argument being the integer variable to use as a pointer to the current object during iteration. If a **loop** command is executed with no parameters, **noun3** is used as the default variable. When the **loop** command is executed, iteration starts with the variable being set to 1 (the first object or location in the game file). When the matching **endloop** command is executed the variable is incremented to point to the next object or location and execution continues from the first line after the original **loop** command. The loop will end when the **endloop** command is executed with the iteration variable already pointing to the last object or location. When the loop ends, execution continues from the first line after the **endloop** command. For example, the following code will output the short description of each object and location as a list:

```
{+print_objects  
loop
```



```

    write noun3{List} ^
endloop
}
```

As another example, this function will output all the children of the object passed as an argument to it, using the variable **POINTER** as the iteration variable:

```

integer POINTER

{+print_children
write "Children of " arg[0]{the} " :^"
loop POINTER
  if POINTER(parent) = arg[0]
    write " " POINTER{the} ^
  endif
endloop
}
```

It is not legal to nest loops unless the inner loop is within a function that is called from within the outer loop. To return out of a loop early you can either return from the currently executing function or set the iteration variable to point to the last object or location manually. For example, to return out of a loop after the fifth object, use the constant **objects** like this:

```

loop
  write noun3{The} ^
  if noun3 = 5
    # SET THE CONTAINER noun3 TO POINT TO THE
    # LAST OBJECT OR LOCATION SO THAT THE
    # LOOP WILL STOP ITERATING
    set noun3 = objects
  endif
endloop
```

The index of the last object or location (which is also the number of object and locations in the game) is stored in the constant **objects**.

The SELECT and ENDSELECT Commands

The **select** command is similar to the **loop** command except that it only iterates over a subset of game's objects. This subset is defined by passing a parameter to the **select** command. This parameter can be one of the following three things:

1. an object that all the selected objects must be a direct child of
2. an attribute that all the selected objects must have, or
3. a scope that all the selected objects must be in

It is possible to prefix any of the above three criterion with a shriek (!) to indicate that the objects to be selected should *not* meet the specified criterion.

Although the **loop** command can be used to produce any list of objects that the **select** command can, the **select** command is significantly faster as the bulk of the filtering is done by compiled code. Even if the selected objects are filtered with further **if** statements as in a regular loop, the number of objects returned by the **select** command is usually a significantly smaller number than the total number of objects in the game, so the remaining processing is relatively quick. Below is an example of the loop above using the **select** command:

```
integer POINTER

{+print_children
write "Children of " arg[0]{the} " :^"
select arg[0] POINTER
  write " " POINTER{the} ^
endselect
}
```

If the desired set of objects cannot be expressed with a single criterion, further **if** statements are placed within the loop such as this code from **verbs.library**:

```
{+contains_liquid?
# CHECK IF THE SPECIFIED OBJECT CONTAINS ANY LIQUIDS
select LIQUID CHILD
  if CHILD(parent) = arg[0]
    return true
  endif
endselect
return false
}
```

Like the **loop** command, if a container is not specified (**CHILD** in the example above), **noun3** is used by default.



Becareful not to use the default iteration container **noun3** more than once when using nested **select** loops. As it is a global variable the loops will interfere with each other if it is shared.

The following scopes can be used with the **select** command:

Scope	Description
*held	All objects held by the player.
*here	All objects in the current location.
*present	All objects either in the current location or currently held.
*anywhere	All objects in the game (use loop command instead).



As only a single parameter can be used with a **select** command it is more efficient to use the criterion that selects the smallest set of the object when **if** statements are required to filter the list further. For example, if you wanted a list of all the objects that are being carried by the player that don't have a **mass** of **scenery**, the it would be more efficient to use the **select** command to find the objects that are children of the player and then **if** statements to check their mass. This is because there will generally be fewer objects in the game that are currently being carried by the player than objects with a **mass** that isn't set to **scenery**. If the **select** command has been used to find all the objects that aren't scenery, the **if** statement to check if each object is also carried by the player would need to be executed many more times.


The REPEAT and UNTIL Commands

A **repeat... until** loop allows you to repeat a section of code until a specified condition is true. The expression or expressions to be tested should follow the **until** statement, using the same syntax as an **if** statement.

The following is an example of a **repeat... until** loop:

```
set INDEX = 10
repeat
  write "DON'T PANIC! "
  set INDEX - 1
until INDEX = 0
```

loop...endloop loops and **repeat...until** loops cannot be nested within a single function. It is legal, however, to place a second loop within a function that is called from within the first loop. For example:



Warning

```
{+print_objects
set INDEX = 10
loop noun3
  write noun3{the} ^
  execute "+print_children"
endloop
}

{+print_children
loop noun4
  if noun3 grandof noun4
    write "    " noun4{the} ^
  endif
endloop
}
```

Note that it is legal, however, to nest a **repeat** loop inside an object **loop** or vice versa.

The WHILE and ENDWHILE Commands

As a **repeat** loop will always happen at least once, if there is any chance that a loop should happen zero times, a **while** loop must be used. A **while** loop performs its test first, then executes the code following the expression if it evaluates to **true**. On reaching an **endwhile** command it will return to the matching **while** command and re-evaluate the expression. When the expression eventually evaluates to **false**, execution will continue from the line after the **endwhile** command. For example, the following function will output the value of the passed arguments:

```
{+arg_values
set INDEX = 0
while INDEX != @arg
  write "Argument" INDEX " = " arg[INDEX] ".^"
  set INDEX + 1
endwhile
}
```

The RETURN Command

return or **return true**

The command **return** will stop execution of the current function and return to the next line after the corresponding **execute** command in the calling function. If the function containing the **return** command was called internally by the JACL interpreter, processing will continue, eventually returning to the command prompt for the player's next move.

return false

The command **return false** will stop execution of the current function just like a **return** command. When it returns, the JACL interpreter will behave as though the function it returned from did not exist, and was therefore not executed at all. This will, in the case of calls originating from **grammar** statements, cause the default action to occur.

The following is a demonstration of the use of **return false**:

```
object bond: james bond

{ask_about_bomb
if bomb(parent) = limbo
    return false ;The function +ask_about will now be
                  ; executed as though this function did
                  ; not exist.
endif
write "~I'm glad you asked...~^"
}
```

This is a common technique in JACL games. Once the player has typed a command that refers to an object, the interpreter will attempt to execute the appropriate function that is associated with that object. For example, if the player was to type **ask bond about the bomb**, the interpreter would attempt to execute the function **ask_about_bomb** that is associated with the object **bond** (the function above). If this function does not exist, the function **+ask_about** will be called instead. When a **return false** command executed, the interpreter will behave as though the local function containing the **return false** does not exist. In essence, this allows a function to override the default result of a verb under some conditions and accept it under others.

Changing Data

It is often desirable to change the value of data that is defined when the game is first loaded. This is achieved using the **set** command for integers, objects and locations (see Typecasting below), and the **setstring** command for text.

The SET Command

Any **set** command must be of the following format:

```
set container operator value [ operator value...]
```

The **set** command enables you to modify the current value of the specified container. A container is either an object element (see the chapter [Definitions in Detail](#) for more information), a variable or an item pointer such as **noun1** or **here**. The *value* can be an integer, integer constant or any other container whose value you wish to copy.

The following is a list of operators that can be used with the **set** command:

Operator	Description
=	Set the value of the specified container to the specified value.
+	Add the specified value to the current value of the specified container.
-	Subtract the specified value from the current value of the specified container.
/	Divide the current value of the specified container by the specified value.
*	Multiply the current value of the specified container by the specified value.
%	The modulo operator calculates the remainder value of the specified container when divided by the specified value.

The following example demonstrates how to add 7 and 8 then divide the result by 5:

```
integer TEMP  
  
{+maths  
set TEMP = 7 + 8 / 5  
}
```

This line of code can be read as a container to apply the operations to followed by groups of two parameters: an operator and a value. In this case the three operations are:

1. make TEMP equal to 7
2. add 8 to TEMP
3. divided TEMP by 5




Warning

Although all JACL commands require whitespace between each parameter, it is especially easy to forget the space between the container and the operator or the operator and the value. It is, of course, valid to have a command such as:

```
set BANK_BALANCE = -42
```

The **set** command also has the ability to use the current value of a string as the name of the container or value being referred to. For example, if an integer variable called **counter** was defined, its value could be set to 1 using the following code:

 **Information**

```
integer counter 0

{+modify
setstring indirection "counter"
set indirection = 1
}
```

This technique can be particularly useful when the need arises to pass a reference to an array as a function argument.

Type Casting

The JACL language allows the definition of complex types such as objects and locations at load time, but once the game is running, the only types that can be modified are integers and strings. When objects and locations are created they are assigned an integer value. The first object or location defined is given the number one, the remainder are numbered sequentially in the order they appear in the source file. For the purposes of numbering there is no distinction between objects and locations. Because of this, it is possible to use an integer variable wherever an object label is expected. For example:

```
integer VARIABLE

{+code
set VARIABLE = small_frog
set noun4 = VARIABLE
set noun4 + 1 ; NOTE: noun4 will now equal the item defined
                ; directly after small_frog in the game file.
set lantern(status) = noun1(parent)
set max_rand = objects
set sword(parent) = random
}
```

This code sets an integer variable to the index of the object **small_frog**. It then sets the internal object pointer (another integer variable) **noun4** to that value. This pointer is then incremented to point to the object or location that is defined directly after the **small_frog**. After that, the current parent of the object pointed to by **noun1** is stored in the **status** element of the object **lantern**. Finally, **max_rand** is set to the constant **objects** (the number of objects and locations in the game), and the parent of the object **sword** is randomly set to one of the object or locations from in the game.

As you can see, all these variables, object elements, object labels, object pointers and constants simply resolve into integers and can be used interchangeably.



The only real potential danger in the above examples is that of setting one of the object pointers to a value less than one, or greater than the internal number of the last object. The index of the last object or location (which is also the number of object and locations in the game) is stored in the constant **objects**.

The SETSTRING and ADDSTRING Commands

The **setstring** command is in many ways similar to the **write** command. Where a **write** command can take a list of parameters and will output the result to the screen, the **setstring** command can take an identical list of parameters and store the result in a **string**. The first parameter of a **setstring** command is a **string** to store the resulting output in. All other parameters constitute the text to be stored. Any previous contents in a string are overwritten by a **setstring** command. Below is an example of the **setstring** command in action:

```
string buffer "empty"

{+some_function
setstring buffer "You are currently in " here{the} " on turn " total_moves "."^"
}
```

The **addstring** command works in a similar way except it does not erase the current contents of the string being written to.

The PADSTRING Command

The **padstring** command takes three parameters. The label of the string to fill, the text to fill the string with and an integer specifying the number of times to copy the text into the string. This command primarily exists to assist with presentation within the status window. For example, to print a blank line in the status window, the text that will be copied is a single space in quotes (" ") and it will be copied **status_width** times. The code below demonstrates creating a string called **status_text** that contains enough spaces to fill a line in the status window:

```
string status_text

{+update_status_window
padstring status_text " " status_width
}
```


Movement

The MOVE Command

The **move** command is used to transport an object within your game world and uses the format:

```
move object to destination
```

The *object* can be any object label, object pointer or the integer index of an object, while the *destination* can be any item label, item pointer or the integer index of an item. In essence, the **move** command will set the parent element of the *object* to be equal to the *destination*. In addition to this, it performs several functions relating to the **mass** of the object that is being moved. For example, the command:

```
move note to box
```

will, in addition to moving the note to be a child of the box, automatically increase the **capacity** property of the object (if any) that the note was in before the **move** command, and decrease the **capacity** property of the box. This reflects the decreased burden on note's previous parent and an increase in the box's contents in accordance with the note's **mass**. The following code is equivalent to the above **move** command:

```
set noun4 = note(parent)
set note(parent) = box
set noun4(capacity) + note(mass)
set box(capacity) - note(mass)
```

As you can see, the **move** command is by far the easier option.



If the **capacity** property of the *destination* object ends up a negative number, the **move** command will still successfully complete. It is therefore wise to test for this before issuing a **move** command. Below is an example of this from the library function **+insert_in**:

Warning

```
if noun2(capacity) < noun1(mass)
  write "There is not enough room in " noun2{the}
  write " for " noun1{the} .^
  set TIME = false
  return
endif
```

The TRAVEL Command

A **travel** command must be of the format:

```
travel direction
```

The **travel** command is used to move the player from one location to another. It simulates the player moving in the specified direction under their own steam. The player can also be moved to a different location by directly setting their **parent** to equal the new location. This, however, does not in perform any tests as to whether this movement is physically possible. The **travel** command is normally only ever used by the direction verbs in the library. One of the directions **north**, **south**, **east**, **west**, **up**, **down**, **in**, **out**, **northeast**, **northwest**, **southeast** or **southwest** must be specified as its single parameter.

The JACL Author's Guide

Before the move actually occurs, two functions are executed. If either of these functions exists, and does not **return false**, the move will be prevented from occurring. The first of these functions is the **movement** function associated with the location the player is attempting to move out of. The second, executed if this first function does not exist or issues a **return false** command, is the global function **+movement**. Before either of these functions are called the integer variable **compass** is set to the direction the player is travelling in and the integer variable **destination** is set to the location the player is travelling into.

For example, if the player was in a location called **hut** and typed the command **go north**, the integer variable **destination** would be set to the location they are attempting to move into, say **clearing**, and the integer variable **compass** would be set to the direction he or she is attempting to move in, in this case **north**. Both of these variables would be set before the function **movement** that is associated with the location **hut** is called.

Two common uses of the movement function associated with a particular location are to replace the standard, "You can't go that way," message or to add some text describing the journey from one location to another. For example:

```
location hut : hut
    short      the "hut"
    north      clearing

{movement
if COMPASS != north
    print:
        The only exit from here is the front door
        to the north.^"
    .
    return true
endif
print:
    You step out into the sun light, taking a
    moment for your eyes to adjust.^
.
return false
}
```

Firstly, this function checks if the player is attempting to go in a direction other than this location's only exit. If this is the case, an appropriate message is displayed and a **return true** command is executed. It is this **return** command that prevents the normal message from being displayed. If the player is attempting to travel north, a short description of the journey is displayed and a **return false** is executed. It is this **return false** command that tells the interpreter that the move should continue as normal.

Another common use of an associated **movement** function is to simply remind the player of the available exits when they attempt to travel in an invalid direction. For example:

```
{movement
if destination = nowhere
    print:
        The only exit from here is to the south.^
    .
    return true
endif
return false
}
```

This function would display the message "The only exit from here is to the south." whenever the player attempt to travel in a direction that is currently set to **nowhere**.

Moving Non-player Characters

The file **utils.library** contains several functions to help simulate the natural movement of objects, such as a non-player character walking around. In addition to moving the object, these functions will display the appropriate message required to announce the movement to the player.

The function called from your game to initiate the movement is **+push_object**. This function is passed two arguments: the object to be moved and the direction to move the object in. The direction passed is one of the following integer constants:

Constant name	Integer value
north	0
south	1
east	2
west	3
northeast	4
northwest	5
southeast	6
southwest	7
up	8
down	9
in	10
out	11

Below is an example of the procedure for making an object move from one location to another:

```
execute +push_object<security_guard<north
```

A message will be displayed to the player informing him or her of the object's movement if the player is in either the location the object is moving out of or the location the object is moving in to. If the location the player is in has the attribute **DARKNESS**, the message "You hear footsteps nearby." is displayed.

The DIR_TO and NPC_TO Commands

The **dir_to** and **npc_to** commands calculate the first direction to move in when travelling from one location to another. The **dir_to** command will calculate a route that only passes through locations that have the attribute **KNOWN**, while the **npc_to** command will use all locations.

Both commands use the following syntax:

```
dir_to <Container> <From Location> <To Location>
```

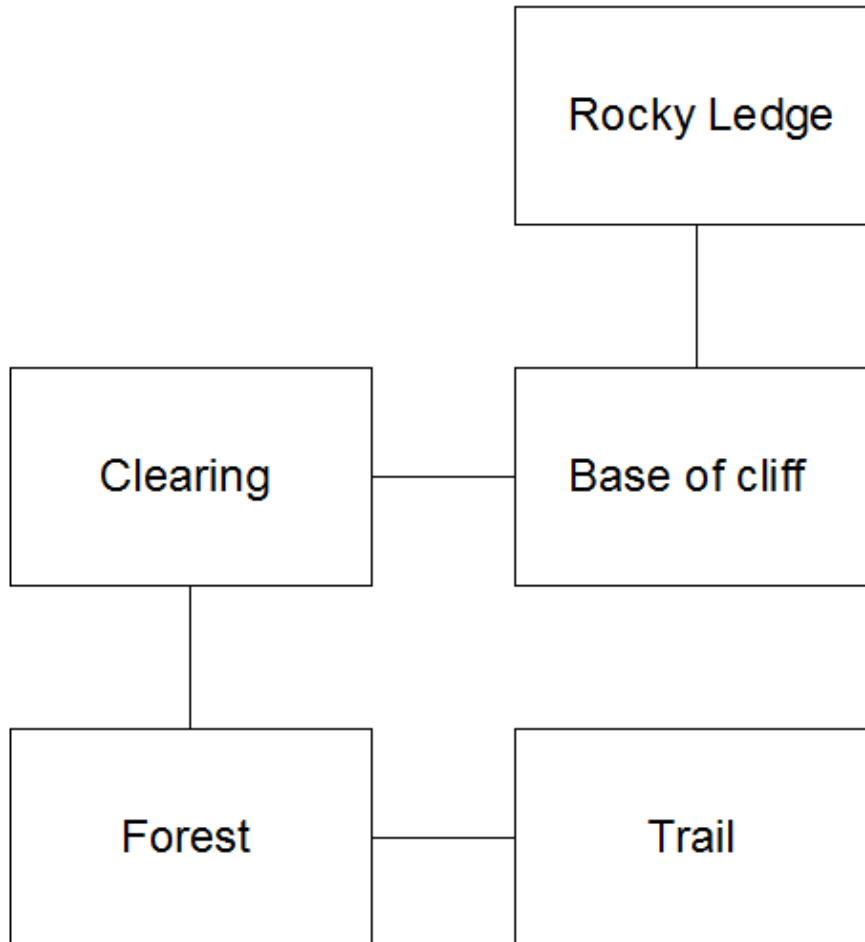
When this command runs, it will store the value of the first direction to move in inside the specified container. This container can be an integer variable or an object element. The value stored is one of the direction

constants from the table above.

For example, given the map below, the command:

```
dir_to INDEX clearing ledge
```

would put the value **north** (0), into the variable INDEX as this is the first direction that must be travelled in from the location **clearing** to get to the location **ledge**.



If the **dir_to** command is unable to calculate a route between the two locations, -1 will be placed into the container.



The **dir_to** command is not able to determine if an exit is only temporarily blocked. If an exit is set to **nowhere** due to a door being closed between the two locations specified, -1 will be returned. If you wish to avoid this limitation you will need to temporarily open all doors that can be opened by the player, calculate the route and then return them to their initial state.

Special-Purpose Commands

The POINTS Command

The **points** command is used to award points to the player and must be followed by a single integer indicating the number of points to be awarded. The awarded points will be added to the integer variable **score**. Below are two examples of the **points** command being used:

```
points 10
points gold(points)
```

When the **points** command is used the following message will be displayed to the player:

```
[YOUR SCORE JUST WENT UP BY n POINTS]
```

The PROXY Command

The **proxy** command is used to issue an in-game move as though it was typed by the player and must be of the following syntax:

proxy *Move*

The specified *Move* may consist of multiple parameters, built up from plain text, variables, constants or macros. When the move is issued by the **proxy** command, all the normal testing will take place and all usual messages will be displayed. The most common use for the **proxy** command is to translate one command into another similar command. For example, sometimes two different commands can be mapped to the same function:

```
grammar put *held in *here    >insert
grammar insert *held in *here >insert
```

With the above two **grammar** statements only the verb is different so mapping them to the same function is possible. In some situations, however, this is not possible as the nouns appear in a different order. Below is an example of how the **proxy** command can save the day:

```
grammar look at *here through *held >look_at_through

{+look_at_through

;main function code tied to this syntax

}

grammar look through *held at *here >look_through_at

{+look_through_at
proxy "look at " noun2{names} " through " noun1{names} ; This will cause
                                                         ; the alternate
                                                         ; syntax to be used
}
```

When the second syntax is used by the player, the only action is to execute a **proxy** command. This issues a command on the player's behalf that matches the first syntax, therefore arriving at the main function with

noun1 and **noun2** pointing to the correct objects.

The macro **{names}** is almost exclusively used with the **proxy** command and outputs all of the names of the object separated by spaces. For example, the following code will issue the command "set safe dial to 42":

```
object dial : safe dial
short      the "safe dial"

{set
execute +build_command<this<42
}

{+issue_set
proxy "set " arg[0]{names} " to " arg[1]
}
```



Don't forget to put spaces before and after any macros or variables used in a **proxy** command.

Trigonometry

JACL provides a set of trigonometric commands to help model a two dimensional space. Each object in JACL has the integer elements **x**, **y**, **bearing** and **velocity**. The values of these elements are read and updated by the following three commands.

The POSITION Command

The **position** command is used to change an object's **x** and **y** elements to simulate it moving on a two-dimensional grid. The movement that occurs depends on the current values for the object's **bearing** and **velocity**. A **position** command is simply followed by the label of the object to be moved:

```
set ship_object(bearing) = 180
set ship_object(velocity) = 100
set ship_object(x) = 500
set ship_object(y) = 500
```

```
position ship_object
```

The above code will set **ship_object(x)** to equal 500 and **ship_object(y)** to equal 400.

The BEARING Command

The **bearing** command is used to calculate the angle from one object to another based on their current **x** and **y** values. A **bearing** command is followed by a variable to store the calculated angle in, the object to measure the angle from, then the object to measure the angle to. For example:

```
bearing INDEX lighthouse ship
write "Bearing from " lighthouse{the} " to "
write ship{the} ": " INDEX " degrees.^^"
```

The above code calculates the bearing from the object **lighthouse** to the object **ship** in degrees (0 - 359), and stores the result in the variable **INDEX**. This calculation is based on the current values of **lighthouse(x)** and **lighthouse(y)** relative to the current values of **ship(x)** and **ship(y)**.

The DISTANCE Command

The **distance** command is used to calculate the distance between two objects based on their current **x** and **y** values. A **distance** command works in the same manner as a **bearing** command:

```
distance INDEX lighthouse ship
write "Distance from " lighthouse{the} " to "
write ship{the} ": " INDEX ^
```

The above code calculates the distance between the object **lighthouse** and the object **ship** and stores the result in the variable **INDEX**. This calculation is based on the current values of **lighthouse(x)** and **lighthouse(y)** relative to the current values of **ship(x)** and **ship(y)**.

The ASKNUMBER AND GETNUMBER Commands

The commands **asknumber** and **getnumber** prompt the player to input a number between a particular range and wait for a response. The only difference between the two commands is that **getnumber** will continue to ask the player for a valid response, only returning when a number between the upper and lower limit is entered. Both of these commands use the following syntax:

asknumber *StorageVariable LowerLimit UpperLimit*

StorageVariable is the integer variable used to store the number entered by the player. *LowerLimit* and *UpperLimit* indicate the inclusive range of numbers that are acceptable. For example, below is some code that prompts the player to enter a number between 1 and 5 (inclusive), and stores the answer in a variable called **RESPONSE**:

```
integer RESPONSE
integer UPPER      5

{+ask_player
...
asknumber RESPONSE 1 UPPER
write "You typed " RESPONSE .^
}
```

When using the **asknumber** command, if the player enters an invalid response, **-1** will be placed in the storage variable and the next line of code will be executed.

The GETSTRING Command

The command **getstring** will prompt the player to enter an arbitrary string of text. The only parameter of a **getstring** is the name of the string variable to store the player's response. Below is an example of the use of the **getstring** command:

```
string players_name

{+intro
```

```
write "What is your name? "
getstring players_name
write "Hello " players_name "!"
}
```

The GETYESORNO Command

The command **getyesorno** will prompt the player to enter **yes** or **no**. It will repeatedly prompt the player to enter a valid response until they do so. The only parameter of a **getyesorno** command is the integer variable used to store the player's response. If the player types **yes** (or **y**), the variable will contain the number 1. If the player types **no** (or **n**), the variable will contain the number 0. Below is an example of the **getyesorno** command in action:

```
constant hints          5

{+ask_player
write "Would you like hints?^"
getyesorno player(hints)
if player(hints) = 1
    write "You typed yes.^"
else
    write "You typed no.^"
endif
}
```

The SAVEGAME and RESTOREGAME Commands

These commands can be used to access the internal functionality to save and restore the current game state. Both of these commands require a container to be passed as a parameter. This container is used to store the return code of the command. If the save or restore operation fails, an appropriate error message will be displayed and the supplied container will be set to **false**. If the save or restore operation is successful, the supplied container will be set to **true** and no message will be displayed. Both commands accept an optional second parameter that is a string to use as the filename. If this parameter is not supplied the Glk library will prompt the player to select a file. A common use of these commands is to define the functions **+save_game** and **+restore_game** to override the internal implementation of this functionality. For example:

```
variable ANSWER
variable RETURN_CODE

{+restore_game
write "Are you sure you wish to restore a saved game?^"
getyesorno ANSWER
if ANSWER = 0
    write "Returning to game.^"
    return
endif
restoregame RETURN_CODE
if RETURN_CODE = true
    print:
        "You cast the spell of time travel and return to the past...^"
    .
endif
}
```

The ability to supply a filename can be used to either allow the player to supply the filename directly after the

save command, or implement a system of auto-saving every tenth move such as this:

```
{+eachturn
set save_timer = total_moves
set save_timer % 10
if save_timer = 0
    set save_count + 1
    setstring filename "autosave"
    addstring filename save_count
    addstring filename ".sav"
    savegame RETURN_CODE filename
endif
```

The TERMINATE Command

The **terminate** command will directly initiate the termination of the JACL interpreter.

The UNDOMOVE Command

The **undomove** command will take the game back to the state it was in prior to the player's last command.

Attributes

Attributes are a set of qualities that an item either **has** or **hasnt**. An item can have none of the possible attributes, all of the possible attributes, or any combination in between. Initial attributes are given to an object in its definition using the keyword **has** (see the chapter [Definitions in Detail](#)), while they are given and taken away during the course of the game using the **ensure** command. Many of the attributes do not have any pre-determined purpose and may be used by the author at will. The best way to determine the full effect of any given attribute is to search for it in the **verbs.library** file.

The ENSURE Command

The **ensure** command is used to change which attributes an item has and uses the following syntax:

```
ensure Item has/hasnt Attribute
```

There is nothing really tricky about the **ensure** command. You are either giving an item an attribute with a command like:

```
ensure dungeon has DARK
```

or taking an attribute away from an item with a command like:

```
ensure lantern hasnt LUMINOUS
```

Object Attributes

Below is a list of the attributes that can be given to an object, and some brief notes on each.

Attribute	Description
CLOSED	If an object has the attribute CLOSED , its contents will not be accessible to the player.
LOCKED	If an object has the attribute LOCKED , it cannot be opened or closed by the player.
DEAD	An object that has the attribute DEAD must also have the attribute ANIMATE . A DEAD object may not be killed, talked to, shown things or given things.
IGNITABLE	An object that has the attribute IGNITABLE may be used to light an object that has the object FLAMMABLE .
WORN	An object that has the attribute WORN will not burden the player with its mass . In order to wear an object it must also have the attribute WEARABLE . While an object has the WORN attribute, the player will not be able to drop it or give it away.
CONCEALING	An object that has the attribute CONCEALING must also have the attribute ANIMATE . This attribute indicates that when the object is examined, the objects that it is carrying should not be listed.
LUMINOUS	When it is present, an object that has the attribute LUMINOUS will allow the player to enter a location that has the attribute DARK .
WEARABLE	The player may wear an object that has the attribute WEARABLE .
CLOSABLE	An object that has the attribute CLOSABLE may be opened and closed by the player.
LOCKABLE	

The JACL Author's Guide

	An object that has the attribute LOCKABLE may be locked and unlocked by the player.
ANIMATE	An object that has the attribute ANIMATE may be killed, talked to, shown things or given things.
LIQUID	An object that has the attribute LIQUID must be carried within an object that has the attribute CONTAINER and may be drunk or poured by the player.
CONTAINER	An object that has the attribute CONTAINER may have other objects put inside it. If it also has CLOSABLE it must not have CLOSED for the player to put things in it. A container may hold as many mass units as its capacity element is set to.
SURFACE	An object that has the attribute SURFACE may have things placed on top of it. A surface may hold as many mass units as its capacity element is set to.
PLURAL	If an object is defined with a plural name, it should be given the attribute PLURAL so the library will avoid printing sentences such as "The boulders is too heavy to go throwing around."
FLAMMABLE	An object that has the attribute FLAMMABLE may be lit using an object that has the attribute IGNITABLE .
BURNING	When an object that has the attribute FLAMMABLE is lit, it will be given the attributes BURNING and LUMINOUS .
ON	Free for use by the author.
DAMAGED	Free for use by the author.
FEMALE	This attribute should be given to any object that has the attribute ANIMATE and represents a female character. This will enable the library to refer to the character correctly.
POSSESSIVE	If an object has the attributes ANIMATE and POSSESSIVE the player will not be able to take objects from them or ask them for objects unless specifically coded for.
OUT_OF_REACH	An object that has the attribute OUT_OF_REACH cannot be touched, even when it is in the same location as the player.
TOUCHED	An object that has the attribute TOUCHED has been moved from its starting position by the player. If the object's long property is set to function , this attribute can be tested to give an appropriate description.
SCORED	Free for use by the author.
SITTING	If the player is currently sitting, this attribute should be given, as the function +movement will check if it is set before moving the player.
NO_TAB	This attribute indicates that the object's names should not be included during tab completion. Tab completion is only used by the console interpreter.
NOT_IMPORTANT	This attribute is given to an object that is of no consequence game. It is usually used to implement nouns that are referred to in a location description but are not part of the game world the player needs to interact with. Verbs in the library will respond with You don't need to worry about that. when the player attempts to interact with an object that has this attribute.

Location Attributes

Below is a list of the attributes that can be given to a location, and some brief notes on each.

Attribute	Description
-----------	-------------

VISITED MAPPED	When the player enters a location, it is automatically given the attributes VISITED and MAPPED after the look function is run. These attributes may be tested for to provide a shorter description on subsequent visits. When the JACL interpreter is set to verbose mode (the DISPLAY_MODE variable is set to 1), locations will have the attribute VISITED taken away from them before the look function is executed.
DARK	If a location has the attribute DARK , the player must be carrying an object that has the attribute LUMINOUS in order to enter it. If the player's current location has the attribute DARK and no object with the attribute LUMINOUS is present, the location will be given the attribute DARKNESS and the player will be prevented from performing any actions that require the ability to see. You should never manually give or take the attribute DARKNESS to or from a location, only the attribute DARK .
DARKNESS	
ON_WATER	If a location has the attribute ON_WATER , any object dropped will sink out of sight and end up in limbo .
UNDER_WATER	If a location has the attribute UNDER_WATER , nothing may be lit, exposed liquids will dissipate and no talking is possible.
WITHOUT_AIR	If a location has the attribute WITHOUT_AIR , the OXYGEN_LEFT variable will be decremented each turn until it reaches zero and suffocation occurs.
OUTDOORS	If a location has the attribute OUTDOORS , any desired effects of weather should be applied.
MID_AIR	If a location has the attribute MID_AIR , any object dropped will fall out of sight and end up in limbo . Actions such as jumping will not be allowed either.
MID_WATER	If a location has the attribute MID_WATER any object dropped will sink out of sight and end up in limbo .
TIGHT_ROPE	If a location has the attribute TIGHT_ROPE , any object dropped will fall out of sight and end up in limbo .
POLLUTED	Free for use by the author.
SOLVED	Free for use by the author.
LOCATION	All locations have the attribute LOCATION . If this is taken away the location will become an object.
SCORED	Free for use by the author.
NO_TAB	This attribute indicates that the location's names should not be included during tab completion. Tab completion is only used by the console interpreter. This attribute is given to a location that is of no consequence game. It is usually used to implement nouns that are referred to in a location description but are not part of the
NOT_IMPORTANT	game world the player needs to interact with. Verbs in the library will respond with You don't need to worry about that. when the player attempts to interact with an object that has this attribute.

User Attributes

A common use of attributes is to check if an action has already been performed. This may be useful if it is not logically possible to perform the action twice, or each subsequent attempt would result in a different outcome. Another possibility is that you would like to give a lengthy response the first time, then a short response on all subsequent times. To help facilitate this, and many other types of game-specific behaviour, JACL allows you to define up to 32 user attributes. User attributes are defined using the **attribute** directive followed by one or more attribute names. Below is a rather minimal example of using user attributes:

The JACL Author's Guide

```
attribute DOOR_OPENED

{open_override
ensure door hasnt CLOSED
if door hasnt DOOR_OPENED
  write "You hold your breath as the door slowly "
  write "creaks open.^\n"
  ensure door has DOOR_OPENED
  return
endif
write "You open the door again.^\n"
}
```

The above code creates a user attributes called **DOOR_OPENED** and gives the object **door** that attribute when it is first opened. It also tests whether the door already has this attribute before deciding which message to display.

Functions

Functions in JACL are similar in principle to functions and procedures in many other programming languages. They act as sub-routines, discrete units of code that can be executed manually from other functions or internally by the interpreter. There are two fundamental types of functions: global and associated. A global function is independent of any object and is designated as such by being given a name beginning with a plus sign. Any function whose name does not begin with a plus sign is automatically associated with the nearest object or location above it in the game file.



It is illegal to define a non-global function before the first object or location as it will not have an object or location to be associated with.

A function begins with an opening curly brace ({) that is followed directly by the name of the function. It is possible for a function to have multiple names by providing additional names on the same line as the opening curly brace, each separated by whitespace. Below is an example of a function associated with an object:

```
object boulder

{take : push : turn
write "The boulder is way too heavy to move.^"
set time = false
}
```

This function has three names. As none of the names begin with a plus sign, they are all names associated with the object **boulder**.

The full internal name of an associated function is constructed by taking the name as it appears in the program, then appending an underscore and the label of the object or location that it is associated with. For example, the above function has the full internal names of: **take_boulder**, **push_boulder** and **turn_boulder**.

If the name of a function begins with a plus sign, it is considered to be a global function and the label of the nearest object is not appended to the supplied name. For example, the function **+eachturn** has the full internal name of **+eachturn**.

The EXECUTE and CALL Commands

All **execute** and **call** commands are of the following format:

```
execute/call [object.]FunctionName[<arg1<arg2...]
```

The **execute** command allows the manual execution of any function by specifying its full internal name. The **call** command is almost identical except it will not display an error if the function does not exist. This behaviour is required when calling a function that contains some optional, extra code that may or may not exist. When executing functions manually using the **execute** or **call** commands, the full internal name must be specified. With global functions, this is simply the name of the function as it appears in the program. For example, the following function:

```
{+hello
write "Hello world!"
}
```

would be called with the command:

```
execute +hello
```

The full internal name of an associated function is constructed by taking the name as it appears in the program, then appending an underscore and the label of the object or location that it is associated with. For example, the following function:

```
object wheel : steering wheel

{examine
write "The steering wheel is covered in black leather.^[^"
}
```

would be called with the command:

```
execute examine_wheel
```

Once a function is executing, the string constant **function_name** is set to contain the full internal name that was used to call the function. This can be useful if the function has multiple names and you need to modify its behaviour based on which function was used to call it. For example:

```
{+intro
execute "+example"
}

{+test : +example : +multi
write function_name ^
}
```

This code will simply output the string **+example**

An **execute** command also allows the name of a function be to prefixed with an item label or pointer, followed by a period. This tells the interpreter to execute the specified function that is associated with the specified item. For example, the above command could also be expressed as:

```
execute wheel.examine
```

The advantage of this syntax is that an item pointer or variable can be used in place of the object label **wheel**. The following code snippet is equivalent to the command above:

```
set noun4 = wheel
execute noun4.examine
```

When using the syntax of an object pointer or label followed by a period, it is legal to supply a variable, integer constant or object element as the function name. This is useful when you wish to either iterate through a series of functions associated with an object or dynamically map an action to a function at run-time. The name of the function called will be the current integer value of the supplied variable. For example, the code below demonstrates two ways to call the function **1** that is associated with the dial:

```
constant          setting 2

object dial : dial
  short          a "dial"
  setting        5
```



```
{1
write "You set the dial to one^"
}

set dial(setting) = 1

execute dial.dial(setting)

# OR, MORE DIRECTLY...
execute 1_dial
```

It is possible to associate a function with more than one object by prefixing the function name with an asterisk (*). When you prefix a function with an asterisk the full internal name of the function will be stored exactly as the name supplied. This allows you to construct a name that mirrors the name that would be created for a normal associated function. For example:

```
object redball : red ball

object blueball : blue ball

object yellowball : yellow ball

{kick : *kick_redball : *kick_blueball
write noun1{The} " sails high in the air.^"
}
```

The above function has three names, the first automatically associating it with the yellow ball in the normally fashion with the second two manually associating it with the objects **redball** and **blueball**. The order the names are defined in is not important.



When using the above technique to manually associate a function with an object, the label of the object must not contain an underscore. This is because the supplied function name is parsed from the right, with everything after the first underscore encountered being considered the object label. If a function was to be given the name ***kick_blue_ball**, the interpreter would attempt to associate the function **kick_blue** with the object **ball**.

It is also possible to supply the name of the function to be executed in a string variable or constant. This technique allows strings to be used as function pointers and is used by the **menu.library** as a way of passing a call-back function to function in the library. See the chapter on the [menu.library](#) for an example.

A function will stop executing and return to the function that it was called from when it encounters a **return** command or arrives at the closing brace. If a function reaches its closing brace, an implicit **return true** is executed.

Passing Arguments to a Function

It is possible to pass string and integer arguments to a function when executing it. This is done by following the function name by a less-than symbol (<) followed by the value to pass. Each additional argument is separated by another less-than symbol. When the specified function is executed, the arrays **arg** and **string_arg** are populated with the values passed. The array **string_arg** stores a copy of the string value of every argument passed. If a string variable or constant is supplied as an argument, the value of the string constant is stored, not the name of the constant itself. The array **arg** stores integer value for every argument that can be resolved to an integer. If an argument can't be resolved to an integer, -1 is stored at that point in the array. The arrays **arg** and **string_arg** are always of equal length, being the total number of arguments

supplied. Below is an example of a function call that passes seven arguments and a function that displays them:

```
string    test    "This is a string constant."

variable DEPTH    0
variable INDEX    0

{+some_function
...
set DEPTH = 99
execute "+subfunction<This is a literal string.<42<Fred<test<12<DEPTH<13"
...
}

{+subfunction
set INDEX = 0
while INDEX != @arg
  write "arg[" INDEX "]: " arg[INDEX] "  string_arg[" INDEX "]: " string_arg[INDEX] ^
  set INDEX + 1
endwhile
}
```

The above code produces the following output:

```
arg[0]: -1    string_arg[0]: This is a literal string.
arg[1]: 42    string_arg[1]: 42
arg[2]: -1    string_arg[2]: Fred
arg[3]: -1    string_arg[3]: This is a string constant.
arg[4]: 12    string_arg[4]: 12
arg[5]: 99    string_arg[5]: fuel_left
arg[6]: 13    string_arg[6]: 13
```



The first argument passed to a function is also stored in the object pointer **noun3** for historical reasons.

The function-call count

Every time a function is executed, an internal count of the number of times it has been called is increased by one. The value of this count is obtained by prefixing the full internal name of the function with an at symbol (@). If an at symbol is used on its own, the number of times the current function has executed is returned.

For example, below is the same example used in the section [User Attributes, modified to use the function-call count](#):

```
{open_override
ensure door hasnt CLOSED
if @ = 1
  write "You hold your breath as the door slowly "
  write "creaks open.^^"
  return
endif
write "You open the door again.^^"
}
```

The RETURN Command

The **return** command is used to pass a value back to the function that called it, or the interpreter if called internally. A **return** command with no parameters will return the value **1**, which is the same as a function simply reaching its closing bracket. If a value is specified as a parameter to a **return** command, that value will be returned instead. For example:

```
{+some_function
set RESULT = +adder<16<21<42<75
write "The result is: " RESULT
}

{+adder
set INDEX = 0
set COUNTER = 0
while INDEX != @arg
  set COUNTER + arg[INDEX]
  set INDEX + 1
endwhile
return COUNTER
}
```

The above function **+adder** will sum all the values passed as arguments and then return the result to the calling function.

Responding to the Player's Moves

In this section you will learn more about the function calls made by the interpreter when the player types a move while playing a game. As the first step in processing the player's move, the interpreter attempts to find a **grammar** statement that matches the command typed. For more information see the section on [Grammar Statements](#). If a match is found, the function after the greater-than symbol at the end of the **grammar** statement is used as the core name for a series of function calls. This mapping of the player's moves to functions through the use of **grammar** statements is one of the fundamental principles of writing a JACL game.

Before calling any functions, the interpreter will set two object pointers, **noun1** and **noun2**. These are set to the objects referred to in the move typed by the player in the order they occur. For example, for a move like "insert card in slot", **noun1** would be set to the card, and **noun2** would be set to the slot. We will start, however, by examining a move that refers to a single object, such as "take wooden pole".

The **grammar** statement that matches the move "take wooden pole" looks like this:

```
grammar take *here >take
```

This **grammar** statement says that if a move consisting of the word **take** followed by an object that is in the current location is typed, execute the function **take**. In reality there are a number of possible functions that can be called, each having **take** as a part of their name. For the purpose of the following examples, we will assume that the object "wooden pole" has the label **pole**.

After the player types the move "take wooden pole", the first function the interpreter will attempt to execute is the global function **+before**. If this function exists, and does not return **false**, no further processing is performed with regard to this move.

The JACL Author's Guide

The next function the interpreter will attempt to execute is the global function **+before_take**. If this function exists, and does not return **false**, no further processing is performed with regard to this move. Below is an example of what this function might look like:

```
{+before_take
if guard is *here
  write "You decided to leave " noun1{the} " alone "
  write "while the guard is around."
  return
endif
return false ;continue as normal
}
```

As you can see, the **+before_take** function is independent of the object being taken. This makes it ideal for situations that affect all objects. If this function returns **false**, or does not exist at all, the interpreter will next attempt to execute the function **take_pole**. This function will appear in the game file as a function called **take** that is associated with the object **pole**.

If this function exists, it will be executed in place of the default global action for the **take** verb. If this function does not exist, or returns **false**, the global function **+take** will be executed. This function contains the default outcome for the **take** verb. Below is the **+take** function from the library:

```
{+take
if +important<noun1 = true
  return true
endif
if +darkness = true
  return true
endif
if +reach<noun1 = true
  return true
endif
if player has SITTING
  write "You will have to stand up first.^[
  set TIME = false
  return
endif
if noun1(mass) >= heavy : noun1 has LOCATION
  execute +move_scenery
  return
endif
if noun1(mass) > player(capacity)
  write "You are carrying too much to take " noun1{the} .^[
  set TIME = false
  return
endif
if noun1 has LIQUID
  write noun1{The} " run" noun1{s} " through your fingers.^[
  return
endif
override
write "You take " noun1{the} .^[
move noun1 to player
ensure noun1 has TOUCHED
}
```

This function performs a few simple tests to confirm the move is possible then moves the object being taken to the player. When this function reaches the **override** command (the fourth line from the end), the interpreter

will attempt to execute the function **take_override_pole**. This will appear in the game file as a function called **take_override** that is associated with the object **pole**. If this function exists, it will replace all the code that comes after the **override** command in the function **+take**. This allows an object-specific outcome to be coded for, while still taking advantage of all the tests that precede the **override** command performed. For this reason, an **override** function is only of use when there is a chance that the player's move may not be possible. This is the case with the **take** command in situations such as when the player is already carrying too much, the object they are attempting to take is out of reach, or it is a liquid.

If the function **take_override_pole** does not exist, the interpreter will attempt to execute the function **+default_take**. This function allows the author to code a default override function that applies to all objects.



The same effect can be achieved by modifying the code after the **override** command of the **+take** function in the library. Putting this code in **+default_take**, however, allows the library to be upgraded to a newer version at any time without losing your game-specific modifications. This is obviously the preferred method.

If the **override** command in the function **+take** is reached, and neither a **take_override_pole** or **+default_take** function exists, execution will continue from the line after the **override** command.

The final stage in processing the player's move calls a series of **after** functions, regardless of the outcome of any preceding it. It is not important whether any **after** function executes a **return** or **return false** command, as all three **after** function will execute in order regardless of outcome of the one before. The first function to be called in this final stage is the local function **after_take** that is associated with the object **pole**. This function provides the opportunity to perform any processing required after the move **take pole** is issued by the player, regardless of any previous outcome.

The next function called is **+after_take**. This function provides the opportunity to display any additional text relevant only to the **take** verb, but independent of any objects referred to. Below is an example of this:

```
{+after_take
if noun1 = cookie
  if cookie(parent) = player
    if fred is *here
      write "Fred looks a bit upset that you have"
      write "taken the last cookie.^^"
      return
    endif
  endif
endif
endif
}
```

Finally the global function **+after** is called. This function provides the opportunity to perform any additional processing before the player's move is complete, regardless of the verb used or any objects referred to.

The above example details the function calls made for a command referring to a single object. The following three lists detail all the functions called for an in-game command containing no objects, one object and two objects respectively.

grammar *verb* >*CoreFunction*

1. The interpreter attempts to execute the function **+before_CoreFunction**. If this function exists and does not return **false**, execution will skip directly to **+after_CoreFunction**.

2. If it does not exist, or returns **false**, an attempt will be made to execute *CoreFunction_CurrentLocation*. This is a function called *CoreFunction* that is associated with the current location.
3. If this does not exist, an attempt will be made to execute the global function *+CoreFunction*.
4. If this function contains an **override** command, an attempt will be made to execute *CoreFunction_override_CurrentLocation*. This is a function called *CoreFunction_override* that is associated with the current location.
5. If it does not exist, or returns **false**, an attempt will be made to execute the function *+default_CoreFunction*.
6. If this does not exist, or returns **false**, execution will continue from the line after the **override** command.
7. The interpreter attempts to execute the function *+after_CoreFunction*.

grammar *verb *Object1 >CoreFunction*

1. The interpreter attempts to execute the function *+before_CoreFunction*. If this function exists and does not return **false**, execution will skip directly to *+after_CoreFunction*.
2. If it does not exist, or returns **false**, an attempt will be made to execute *CoreFunction_Object1*. This is a function called *CoreFunction* that is associated with *Object1*.
3. If this does not exist, an attempt will be made to execute the global function *+CoreFunction*.
4. If this function contains an **override** command, an attempt will be made to execute *CoreFunction_override_Object1*. This is a function called *CoreFunction_override* that is associated with the specified object.
5. If it does not exist, or returns **false**, an attempt will be made to execute the function *+default_CoreFunction*.
6. If this does not exist, or returns **false**, execution will continue from the line after the **override** command.
7. The interpreter attempts to execute the function *+after_CoreFunction*.

grammar *verb *Object1 preposition *Object2 >CoreFunction*

1. The interpreter attempts to execute the function *+before_CoreFunction*. If this function exists and does not return **false**, execution will skip directly to *+after_CoreFunction*.
2. If it does not exist, or returns **false**, an attempt will be made to execute *CoreFunction_Object2_Object1*. This is a function called *CoreFunction_Object2* that is associated with *Object1*.
3. If this does not exist, or returns **false**, an attempt will be made to execute the global function *+CoreFunction*.
4. If this function contains an **override** command, an attempt will be made to execute *CoreFunction_Object2_override_Object1*. This is a function called *CoreFunction_Object2_override* that is associated with *Object1*.
5. If it does not exist, an attempt will be made to execute the function *+default_CoreFunction*.
6. If this does not exist, or returns **false**, execution will continue from the line after the **override** command.
7. The interpreter attempts to execute the function *+after_CoreFunction*.

Special Functions

The following are some special purpose functions that are called internally by the JACL interpreter:

Function	Description
+bootstrap	This function is only executed once when a game first loads. Any initialisation code that must be run before the +header function is executed must go here.
+intro	This function is executed when a game is first run or restarted. It is used to display introductory text and set the starting values for any variables required.
+header	This function is the very first to be executed before the player's command is processed when playing with a CGI interpreter.
+footer	This function is the very last to be executed after the player's command has been processed when playing with a CGI interpreter.
+top	This function is the very first to be executed before the player's command is processed when playing with a GLK interpreter.
+bottom	This function is the very last to be executed after the player's command has been processed when playing with a GLK interpreter.
eachturn_here	If the current location has an eachturn function associated with it, it will be executed directly before, and under the same conditions as, +eachturn .
+eachturn	This function is executed each time a successful command is entered by the player. The interpreter decides on whether or not a command was successful by examining the state of the variable TIME . If it is set to true , the +eachturn function will be executed (and the TOTAL_MOVES variable will be incremented by one), just before +footer is executed.
+system_eachturn	This is the final function to be executed after each successful command is entered by the player. This function is used to execute library code that is not game-specific and must be run after each of the player's commands.
+dark_description	This function is called by the interpreter if a look command is executed in a location that has the attribute DARKNESS .
+object_descriptions	This function is called by the interpreter as the last step in processing a look command. It must display text that indicates the presence of all objects in the current location that don't have a mass of scenery .
+no_light	This function is called by verbs in the library if they are attempted by the player in a location that has the attribute DARKNESS .
+movement	This function is executed each time the player attempts to move to another location. If this function returns false (it does not exist or exited with the command return false), then the player's attempted movement is successful. If it does exist and does not exit with the command return false (reaches the end of the function or executes a return (return true) command), then the player is not moved. In this case, some text explaining why the movement did not occur should be displayed.
movement_here	If the current location has a local movement function associated with it, it will be executed directly before, and under the same conditions as, +movement .
+before_look	This is the first function executed whenever a look command is executed. If it returns true no further processing of the look command occurs.
+title	This function is executed after +before_look , but before the locations associated look function. This function is the place to put any generic code that prints the title of each location, or extra meta information such as whether the player is currently sitting down.
look_here	This function is executed whenever the player types a look command, moves into a new location or restores as saved game.
+object_descriptions	

This function is executed after the above **look** function to output the descriptions of all the objects present in the current location.

+after_look	This is the last function executed whenever a look command is executed.
constructor_item	This function is executed for each item defined straight after the game file is loaded and before +intro is executed.
+save_game	These functions may be defined to override the internal implementation of the respective system-level commands. When the player attempts to use one of these commands, the interpreter will first look for the presence of the corresponding global function. If this function exists it will be executed. If it does not exist, the default implementation inside the interpreter will be used.
+restore_game	
+restart_game	
+undo_move	
+quit_game	

Utility Functions

The following are utility functions that are provided by **utils.library**:

Function	Description
+no_light	This function is called by verbs in the verbs.library if the player attempts to use them in a location that has the attribute DARKNESS .
+details <i>Object</i>	This function displays information about the object that is passed to it as a parameter. This information includes whether the object is open or closed and any other objects that are contained within or being carried by this object.
+contents <i>Object</i>	This function displays a list of any other objects that are contained within or being carried by the object passed to it as a parameter. This function is called by +details .
+spaced_contents <i>Object</i>	This function is similar to +contents except that it starts a new paragraph if there are any objects to list. It is more suited for use after location descriptions.

Creating New Verbs

Although the implementation of a large number of standard verbs is available in **verbs.library**, almost all large games will find the need to define custom verbs. New verbs are defined using **grammar** statements and their default implementation is provided in a global function. This chapter contains information on making your custom verbs as robust and complete as possible.

When adding a new verb, it is important to be sure that you are doing a good thing. Adding a new verb simply to facilitate a guess-the-word type puzzle is definitely a bad thing. On the other hand, having an obvious verb missing is almost as annoying for the player as being made to guess an obscure one. Also consider that adding a new verb doesn't always mean adding a new function. You may find that a required verb is simply a synonym for an existing verb. In this case, simply add a new **grammar** statement with the synonym and point it to an existing function. In the library file you will find many functions that are mapped to from more than one **grammar** statement. For example, here is the start of the global function **+insert** and its matching **grammar** statements:

```
grammar insert *held on *present      >insert_on
grammar put  *held on *present        >insert_on

{+insert_on
if +reach<noun2 = true
return true
endif
...
}
```

Another possibility is to define "put" as an actual **synonym** of "insert". There are potential dangers with this method that are described in the section on [Synonyms](#).

Now for a few general rules. Each verb that involves touching an object should check that the object does not have the attribute **OUT_OF_REACH** before allowing the player to manipulate it. This, of course, does not apply to verbs that can only be performed on objects that are being held. The following line of code is an example from the top of the **+take** function:

```
{+take
...
if +reach<noun1 = true
    return true
endif
...
}
```

These above lines tell the JACL interpreter to return **true** if the function **+reach** returns **true**. In practice, this means that if the object is unreachable by the player, nothing beyond this line will be executed. Below is the content of the **+reach** function from **verbs.library**:

```
{+reach
if arg[0] has OUT_OF_REACH
    write arg[0]{The} " " arg[0]{is} " out of reach.^"
    set time = false
    return true
endif
return false
}
```

The JACL Author's Guide

This function simply tests if the specified object has the attribute **OUT_OF_REACH**. If so, an appropriate message is displayed, the variable **time** is set to **false** and the function returns **true**. If the object does not have the attribute **OUT_OF_REACH**, the function will return **false**.

Most verbs will also make a similar call to the function **+darkness**. This function tests whether the player is currently in darkness or not. Actions that require the player to see should have the lines:

```
if +darkness = true
    return true
endif
```

Finally, all verbs should contain a code block that calls the function **+important**. This function tests whether the passed object has the attribute **NOT_IMPORTANT** and displays a suitable **You don't have to worry about that.** type message. If the verb you are adding refers to two objects, you may need to call this function twice, one for each object. If one of the objects in the command needs to be an object that is ***held**, you will only need to call **+important** for the other object, as objects with the attribute **NOT_IMPORTANT** can't be taken due to the **take** verb stopping when it reaches this same test. Below is an example of a call to **+important**:

```
if +important<noun1 = true
    return true
endif
```

If your verb causes the object to be moved, such as the **take** verb, you must also ensure that the object is given the attribute **TOUCHED** if the move was successful. This will ensure that any tests as to whether the object has been moved from its initial position or not will be accurate. This time an example from the end of the **+take** function:

```
...
override
write "You take " noun1{the} .^
move noun1 to player
ensure noun1 has TOUCHED
}
```

If the verb performs any tests to check whether the move should be successfully completed under the current circumstances or not, an **override** command should be added directly before any effects are coded, such as in the example above. This allows **override** functions to be associated with objects in order to change the default outcome while still taking advantage of the tests you have coded. Below is an example of the types of tests that the default action for a verb should perform. Of course, the exact tests you will require are specific to the nature of the verb you are coding.

```
grammar ask *present for *carried                >ask_for

{+ask_for
if here has UNDER_WATER
    write "Talking under water isn't very easy.^.^"
    set TIME = false
    return
endif
if noun1 hasnt ANIMATE
    write noun1{The} " seem" noun1{s} " to be ignoring "
    write "your request.^.^"
    return
endif
if noun1 has DEAD
```

The JACL Author's Guide

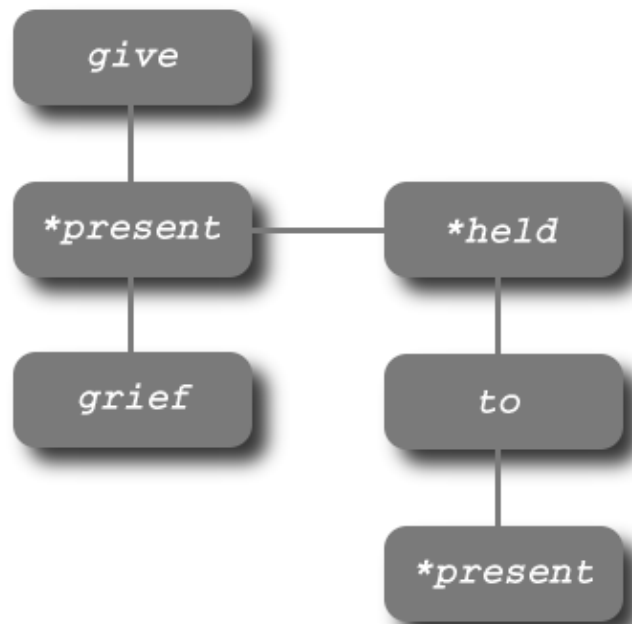
```
write noun1{The} " " noun1{is} " a bit too dead to "  
write "respond.^^"  
set TIME = false  
return  
endif  
if noun1 = player  
write "I think it might be time to take a break and "  
write "get a cup of tea.^^"  
set TIME = false  
return  
endif
```

As you will have seen above, it is also important that the variable **TIME** is set to **false** if the move typed by the player could not be performed. Setting **TIME** to **false** tells the interpreter that the **eachturn** functions should not be executed. In other words, if the player's move is not possible, time should not pass.

One last thing to keep in mind when adding **grammar** statements is whether its scope indicators are going to clash with any other existing **grammar** statements. For example, consider the following two lines:

```
grammar give *present grief >hassle  
  
grammar give *held to *present >give_to
```

The vocabulary for each game is stored as a tree. The following tree is a representation of the two **grammar** statements above.



Given the two **grammar** statements above, the command

```
give sword to troll
```

would produce the message

The JACL Author's Guide

You can't use the word "to" in that context.

This is because **sword** is a match for ***present**, therefore causing the parser to branch down a path that only allows to word **grief** to be used next. This is obviously not the desired effect. Using **\$text** in a **grammar** statement is equally dangerous. As an extreme example, using **\$text** as the first word of the first **grammar** statement will prevent any others from ever matching. It is therefore important to bare in mind that **grammar** statements are tested in the order they appear in the game file, starting from the top. To have a **grammar** statement checked last, add it to your game file after the line including **verbs.library**.

Pointers

Wherever an object or a location label can be used, so too can a number of special purpose pointers. When a command referring to a pointer is processed, the pointer will be substituted with the label of the item that it currently points to. This avoids the repetition of code and the associated problems of code-bloat and inconsistency. Below is a function demonstrating the use of pointers:

```
{+where_is
loop noun3
  if noun3 hasnt LOCATION
    noun4 = noun3(parent)
    write noun3{The}
    write " is in " noun4{the} ^
  endif
endloop
}
```

The above function loops through all the objects defined in the game and displays their current whereabouts. The **loop** command sets the object pointer **noun3** to each of the objects in turn, while the pointer **noun4** is manually set to each object's parent.



Internally, an object pointer is nothing more than an integer variable, and any integer variable, constant or element can be set to an object's label and then used anywhere an object is expected.

The two tables below detail each of the internal object and location pointers:

Object Pointers

Pointer	Description
player	This pointer is set to the object that currently represents the player in the game.
noun1	This pointer is set to the first object referred to in the player's last move.
noun2	This pointer is set to the second object referred to in the player's last move.
self or this	This pointer is set to the object that the currently executing function is associated with.

Location Pointers

Pointer	Description
here	This pointer represents the location that the player is currently in. It is synonymous to the element player(parent) .
destination	This represents the location that the player is attempting to move into. The value of destination can be tested in the +movement function before the move is completed. Once the move is complete, destination will equal here until the next move is attempted. This is a read-only pointer and therefore cannot be used as the <i>container</i> parameter of a set command.
self or this	This pointer is set to the location that the currently executing function is associated with.

The JACL Author's Guide



Be sure your code never makes use of a pointer in place of an object or location while it does not point to a valid object or location. A pointer is not pointing to a valid object or location if it is set to a number less than one or greater than the number of objects and locations in the game.

Object Resolution

A large part of playing interactive fiction involves referring to the objects and locations that make up the game's world. The information in this chapter will assist you in coding your game in such a way that the JACL parser will select the object the player intended as often as possible. All references to objects made by the player are parsed for the following possible structure:

```
objects [from other objects] [except objects [from other objects]]
```

The JACL parser will build two lists of objects for any reference made by the player. The first will contain all the objects for inclusion while the second will contain all the objects referred to after the word **except** for exclusion. Once the two lists have been built, the second list will be subtracted from the first list. Both lists can make multiple use of the word **from** to refer to objects, however, the word **except** can not appear more than once.

Object Naming

When the player refers to an object during the course of the game they may use as many or as few of the object's names as are required to uniquely identify it. For example, consider the following three objects:

```
object ball_1: ball
object ball_2: small red ball
object ball_3: big red ball
```

Presuming all these objects were in the current location, if the player referred to **ball**, then **ball_1** would be selected. This is because the JACL interpreter divides the number of names an object has by the number of names supplied to come up with a best match. Although all three objects have the name **ball**, **ball_1** matches 100% while **ball_2** and **ball_3** only match 33%. If the player referred to **red ball**, then **ball_1** would automatically be excluded, as it does not have the name red. On the other hand, both **ball_2** and **ball_3** have the names red ball, and both would match 66%. In this case, a message would be displayed stating that the reference was ambiguous. If the player referred to **big**, then **ball_3** would automatically be selected as neither of the other two objects have the name **big** at all.

It is important to be aware of the number of names that you give each object, and how they relate to each other. It is generally best to have two similar objects have the same number of names, although in some cases you may wish to nominate one to be the default by giving it less names.

For example, in **The Unholy Grail** there are two objects: **beige agar** and **brown agar**. To avoid problems when looking up agar in the encyclopedia, a third object named simply agar was added. This object stays permanently in **limbo** and therefore does not affect any physical manipulation of the two real agar objects. It does, however, get selected when using commands such as **ask_about** that accept objects that are ***anywhere**.

It also possible to define plural names for an object. When the player uses a plural name all the objects that have that plural name will be selected. For example, the above objects could all be given the plural name **balls** using the following code:

```
object ball_1: ball
    plural          balls
object ball_2: small red ball
    plural          balls
object ball_3: big red ball
```

plural

balls

If during the game the player was to type the command **take balls**, all of the above objects would be selected and three separate take commands would be issued. It is the use of the plural name that tells the parser that a reference to multiple objects was intended and that the reference is not ambiguous. It is also possible to further qualify a reference to a plural name by using one or more regular name. For example, the command **take red balls** would result in only the small red ball and the large red ball being taken.

Disambiguation

It is also possible for each object to have the associated function **disambiguate** to assist in the process of disambiguation. The parser will attempt to call this function on every object that is a possible result for an ambiguous reference that needs to be resolved to a single object.

When called, this function will be passed a single argument being the context in which the ambiguous reference was made. The possible values for this argument are:

Value	Description
0	The list of objects for noun1 .
1	The list of objects for noun2 .
2	Exceptions to the list of objects for noun1 .
3	Exceptions to the list of objects for noun2 .
4	The list of objects noun1 should be from.
5	The list of objects noun2 should be from.
6	The list of objects noun1 's exceptions should be from.
7	The list of objects noun2 's exceptions should be from.

To demonstrate, consider a game that had two objects, a red ball and a red book. The following **disambiguate** function could be added to the book in order to allow the commands **read red** and **look in red** to prefer the book to the ball instead of displaying the usual "ambiguous reference" message:

```
object red_book : red book

{disambiguate
if arg = 0
  ifstring command = "read"
    return true
  endif
  ifstringall command[0] = "look" : command[1] = "in"
    return true
  endif
endif
return false
}
```

This function makes use of the **command** array to look at the words used so far in the player's command. Unfortunately as the command has not been fully processed the single resultant action is not yet known. This function communicates its decision to the parser via its return code. If this function returns **false** (or doesn't exist), the object will be included in the list of possible objects and processing will continue as normal. If this function returns **true** this object will be instantly selected as the object the player was referring to. If this function returns **-1** this object will be excluded from contention and processing will continue as normal. If all objects exclude themselves from contention the first will be selected.

The JACL Author's Guide

It is important to keep in mind that this function is only called when an ambiguous reference is made. The command **read ball** will still attempt to read the ball, it is only a command like **read red** that results in an ambiguous reference that will cause these functions to be called. It is also advisable to use this facility sparingly as it departs from the regular disambiguation rules and may lead to confusion for the player. Used appropriately, however, it can help to solve a frustrating disambiguation problem that has been identified during play testing.

Definitions in Detail

A JACL game file consists of two fundamental components: code and data. This chapter focuses on the data and provides a detailed break down of all the possible elements that may be defined. In the case of objects and locations, all their associated properties are also listed.


Objects

```
object Label : [Name1 Name2 Name3...]
  plural      Name1 [Name2 Name3 Name4...]
  has         Attribute1 [Attribute2 Attribute3...]
  short       IndefiniteArticle ShortDescription
  definite    DefiniteArticle
  long        LongDescription / function
  parent      ItemLabel
  mass        Integer / heavy / scenery
  capacity    Integer
  player
  static
```

Each object definition begins with the keyword **object** followed by the object's label. This label is a unique name by which the object will be referred to by any code within the program. The object's label is then followed by a space-delimited list of names. You may specify as many names as you can fit into a single line of JACL code, and if you do not specify any names, the object's label will be set as its one and only name. These are the names the player will use to refer to this object during the course of the game. For more information on object names, see the chapter on [Object Resolution](#).

Following this header are any properties you wish to specify that pertain to the object that you are currently defining. Below is a description of all the possible object properties:

A **plural** keyword must be followed by a space-delimited list of one or more names that the player can use to refer to this object and others like it as a group.

 Be sure not to confuse this keyword with the attribute **PLURAL**. The attribute **PLURAL** is given to any object that, by itself, is considered to be plural. An example of this would be an object that represents a bunch of flowers. Objects that have **plural** names, on the other hand, may be singular objects, like a coin. If a coin is also given the name **coins** using the **plural** keyword, it can then be referred to as a group along with other objects that also have the plural name **coins**.

If, for example, a game has two coins defined, a silver coin and a gold coin, and the player types **take coin**, the parser will ask the player which coin they are referring to. If each of the coins was given the plural name **coins**, and the player typed **take coins**, the parser would issue the take command for both coins. In a situation where there is more than one gold coin and more than one silver coin, this plural name can also be qualified by one of the object's other names. This is done with a command like **take gold coins**. This will cause the parser to issue the take command for all the objects present that have the plural name **coins** and also have the regular name **gold**. If the player had typed **take gold**, the parser would have asked the player which gold coin they were referring to, as without a plural name being used it assumes the intent was to refer to a single object.

A **has** keyword must be followed by a space-delimited list of attributes that the object is to have when the game begins. For more information, see the chapter on [Attributes](#). If this property is omitted the object will start with no attributes.

A **short** property must be followed by two parameters: the object's indefinite article and short description. The *ShortDescription* text, prefixed by the specified *IndefiniteArticle*, is displayed using the object's **{list}** macro in conjunction with a **write** command. When using the object's **{the}** macro, the *ShortDescription* text will normally be displayed prefixed with the word **the** or the specified definite article. If the word **name** is specified as the *IndefiniteArticle*, the *ShortDescription* text is not prefixed with anything, such as in the case of a proper noun. This applies to both the **{list}** and **{the}** macros. If the *ShortDescription* text is plural, the object should be given the attribute **PLURAL** to ensure that it is referred to appropriately by code in the library. If the **short** property is omitted, the *IndefiniteArticle* will be set to **the** and the *ShortDescription* will be set to the object's label.

A **definite** property is used to override the default of **the** that is output when using the object's **{the}** macro. This is only of use when writing games in languages other than English that have gender for inanimate objects.

A **long** property is followed by the text to be displayed when the object is in the current location and the player types a **look** command. If this property is omitted, the object's label is used.



If the text following a **long** property is the word **function**, the function **long** that is associated with this object will be executed whenever the long description text should be displayed. This provides the ability to have lengthy or dynamic descriptions.

A **parent** property must be followed by either a location label (indicating that the object begins the game in that location), or an object label (indicating that the object begins the game within, on top of or being carried by that object). If the object's parent is set to **here**, or the **parent** keyword is omitted altogether, then the object will start in the nearest location defined above it in the game file.

A **mass** property indicates the physical bulk of the object. It must be followed by an integer, the word **heavy** or the word **scenery**. An integer indicates exactly how much the object encumbers the player or fills a container (see the **capacity** property below for further information). The word **scenery** indicates that the object is immovable and that the interpreter should not display its long description after the location description. For this reason, no **long** property is required for object with a mass of **scenery**. The word **heavy** indicates that the object is immovable, but should have the text following its **long** description printed after the **look** function that is associated with its parent location is executed. If this property is omitted, the **mass** for the object is set to **scenery**.



Behind the scenes, a **mass** of **heavy** translates to 99 and a **mass** of **scenery** translates to 100. It is important to keep this in mind if you change the **capacity** property for the player or create any containers.

A **capacity** keyword must be followed by an integer, indicating the number of **mass** units an object with the attribute **CONTAINER**, **SURFACE** or **ANIMATE** can hold. If this property is omitted, the object will have a **capacity** of zero.



Due to this property's default, any object that has the attribute **CONTAINER**, **SURFACE** or **ANIMATE** must also have a **capacity** property that is set to a suitably large value in order to allow an object to accept other objects. The exception to this rule is if the object being given or inserted has a **mass** of 0, but this value should only be used for insignificantly small objects.



In the file **frame.jacl** there is an object with the label **kryten** that is set up to represent the player. This item has a **capacity** of 42. If left unchanged, the player can not simultaneously carry objects whose **mass** properties total more than 42. This figure of 42 should be used as a guide when setting the **capacity** property of other characters, containers or surfaces and the **mass** of takeable objects.

A **player** property has no parameters and indicates that this object is to represent the player in the game. Behind the scenes this sets the object pointer **player** to point to this object. The value of this pointer can be changed during the course of the game if required.

The properties **bearing**, **velocity**, **x** and **y** are used by the special-purpose commands **position**, **bearing** and **distance**. See the chapter on [Special-Purpose Commands](#) for more information.

The properties **next**, **previous**, **child**, **index**, **status**, **state**, **counter**, **points** and **class** have no pre-determined meaning for objects. You are free to set and test these values as required.

When a command requires a numerical value as a parameter, the following object elements can be referred to:

<i>object_label</i> (parent)	0	<i>object_label</i> (index)	8
<i>object_label</i> (capacity)	1	<i>object_label</i> (status)	9
<i>object_label</i> (mass)	2	<i>object_label</i> (state)	10
<i>object_label</i> (bearing)	3	<i>object_label</i> (counter)	11
<i>object_label</i> (velocity)	4	<i>object_label</i> (points)	12
<i>object_label</i> (next)	5	<i>object_label</i> (class)	13
<i>object_label</i> (previous)	6	<i>object_label</i> (x)	14
<i>object_label</i> (child)	7	<i>object_label</i> (y)	15

These elements can be referred to by name or index number. For example, the following two commands are equivalent:

```
set noun4(parent) = chest
set noun4(10) = chest

# ...and to iterate through all properties
set INDEX = 0
repeat
  write "PROPERTY " INDEX ": " noun4(INDEX)
  set INDEX + 1
until INDEX = 16
```

If an object element is to be given a game-specific use, it is often wise to define a constant that describes its use. For example:

The JACL Author's Guide

```
constant fuel_left 11      # 11 is the index of 'counter'

{+accelerate
  set space_ship(fuel_left) - 1
  ...
}
```

These constants can also be used to set the initial state of an object's properties. For example:

```
constant fuel_left 11      # 11 is the index of 'counter'

object space_ship : space ship
  fuel_left      20
```

This code will set the element **space_ship(11)** to equal 20.

Locations

```
location Label : [Name1 Name2 Name3...]
  has      Attribute1 [Attribute2 Attribute3...]
  short    IndefiniteArticle ShortDescription
  definite DefiniteArticle
  north    LocationLabel / nowhere
  northeast LocationLabel / nowhere
  east     LocationLabel / nowhere
  southeast LocationLabel / nowhere
  south    LocationLabel / nowhere
  southwest LocationLabel / nowhere
  west     LocationLabel / nowhere
  northwest LocationLabel / nowhere
  up       LocationLabel / nowhere
  down     LocationLabel / nowhere
  in       LocationLabel / nowhere
  out      LocationLabel / nowhere
  static
```

Each location definition begins with the keyword **location** followed by the location's label. This label is a unique name by which the location will be referred to by any code within the game file. The location's label is then followed by the location's space-delimited list of names. You may specify as many names as you can fit into a single line of JACL code, and if you do not specify any, the location's label will be set as its one and only name. These are the names the player will use to refer to this location during the course of the game. For more information, see the chapter on [Object Resolution](#).

Following this header are any properties you wish to specify that pertain to the location you are currently defining. Below is a description of all the possible location properties:

A **has** keyword must be followed by a space-delimited list of attributes that the location is to have when the game begins. For more information, see the chapter on [Attributes](#). If this property is omitted, the only attribute the location will have when the game is started is **LOCATION**.



Internally, objects and locations are both stored using the same data-structure. In fact, once the game is running, the only difference between the two is that a location has the attribute **LOCATION**.

A **short** property must be followed by two parameters: the location's indefinite article and short description. The *ShortDescription* text, prefixed by the specified *IndefiniteArticle*, is displayed using the location's **{list}** macro in conjunction with a **write** command. When using the location's **{the}** macro, the *ShortDescription* text will normally be displayed prefixed with the word **the** or the specified definite article. If the word **name** is specified as the *IndefiniteArticle*, the *ShortDescription* text is not prefixed with anything, such as in the case of a proper noun. This applies to both the **{list}** and **{the}** macros. If the *ShortDescription* text is plural, the location should be given the attribute **PLURAL** to ensure the it is referred to appropriately by code in the library. If the **short** property is omitted, the *IndefiniteArticle* will be set to **the** and the *ShortDescription* will be set to the location's label.

The directions the player can travel in from this location are defined by the properties **north, northeast, northwest, south, southeast, southwest, east, west, up, down, in** and **out**. A direction property must be followed by the label of the location that the direction leads to when the game is started. The links between locations may be modified during the course of the game to reflect doors opening etc. The constant **nowhere** (0), may be used in place of a location label to indicate that the player may not move in that direction. If a direction is not listed, **nowhere** is the default.

Following each completed location definition should be an associated function called **look**. This function will be executed every time the description for the location is due to be displayed.

The properties **points** and **class** have no pre-determined meaning for locations. You are free to set and test these values as required.

The JACL Author's Guide

When a command requires a numerical value as a parameter, the following location elements can be referred to:

<i>location_label(north)</i>	0	<i>location_label(up)</i>	8
<i>location_label(south)</i>	1	<i>location_label(down)</i>	9
<i>location_label(east)</i>	2	<i>location_label(in)</i>	10
<i>location_label(west)</i>	3	<i>location_label(out)</i>	11
<i>location_label(northeast)</i>	4	<i>location_label(points)</i>	12
<i>location_label(northwest)</i>	5	<i>location_label(class)</i>	13
<i>location_label(southeast)</i>	6	<i>location_label(x)</i>	14
<i>location_label(southwest)</i>	7	<i>location_label(y)</i>	15

These elements can be referred to by name or number. For example, the following two commands are equivalent:

```
set noun4(west) = beach
set noun4(3) = beach

# ...or iterating across all directions to trap
# the player in the current location
set INDEX = 0
repeat
  set here(INDEX) = nowhere
  set INDEX + 1
until INDEX = 12
```

Integer Variables

Integer variables are defined using the keyword **integer** followed by the name of the variable. The starting value for the variable can be set by following the name of the variable with an integer or a previously defined constant. If no value is specified on definition, the variable is initialised with a value of zero.

The following are some examples of variable definitions:

```
constant DEFAULT_POWER      42

integer AIR_LEFT             100
integer LAGERS_DRUNK         ; Set to zero by default
integer AIRLOCK_SEALED       true
integer MOTOR_POWER          DEFAULT_POWER
```




Like all other data definitions, variables can not be defined within the body of a function.

Internal Integer Variables

The following is a list of integer variables defined internally by the JACL interpreter.

Variable	Description
compass	This variable is used to store the direction the player moved in when they travel between locations. This variable may be tested in either of the movement functions allowing you to prevent the move from occurring or displaying some special text as required. The direction travelled is encoded as an integer that can be compared to a set of constants. See the section on <u>Moving Non-player Characters</u> for details.
total_moves	This variable records the number of successful moves entered by the player so far. This variable starts at 0 and is incremented each time a valid command is entered by the player. This is indicated by the value of the variable TIME (see below).
score	This variable indicates how many points the player has scored during the course of the game.
display_mode	This variable indicates whether the interpreter is in verbose or brief mode, with a value of 1 being verbose and 0 being brief . The starting value is 0. In brief mode, each location is given the attribute VISITED when the player enters it, this is not the case when in verbose mode.
internal_version	This variable is set to the major version number of the JACL interpreter that you are using. This can be tested for in the +intro function to ensure compatibility between your game and the version of the interpreter being used to play it.
time	This variable is set to true when the player makes a move. If at no time during the processing of that move it is set to false , two things will happen. Firstly, the variable TOTAL_MOVES will be incremented by one. Secondly, the eachturn functions will be executed if they exist.
max_rand	When the word random is supplied as a parameter to a command expecting an integer value, a random number between one and the current value of max_rand will be generated. The default value is 100.
notify	When set to true the player will be notified of any increase in their score when the points command is used.

String Variables

String variables are defined using the keyword **string** in the same format as an **integer**. For example:

```
string menu_title      "Options:"
```

Once defined, the value of a string can be output by passing the name of the variable as a parameter to a **write** statement. Strings have a maximum length of 256 bytes.

Arrays

It is possible to define more than one **integer** or **string** with the same name, thereby creating an array of values. Arrays can also be created by supplying more than one value during a single declaration. For example:

```
integer FIBONACCI 0 1 1 2 3 5 8 13 21 34 55 89 144
integer FIBONACCI 233 377 610 987 1597 2584 4181
```

The above code will create a single array of variables called **FIBONACCI** that contains 20 values. The number of values held by an array is fixed when the game is first executed and is referenced by using an at symbol (@) followed by the name of the array. Individual elements of an array are accessed by directly following the name of the array with a set of square brackets ([]) containing the index of the element. The first element of an array is at index 0. The following code displays the contents of the **FIBONACCI** array:

```
integer INDEX

{+display_fibonacci
set INDEX = 0

repeat
    write FIBONACCI[INDEX] ^
    set INDEX + 1
until INDEX = @FIBONACCI
}
```



In the above code, **@FIBONACCI** returns the number of elements in the array, which is one greater than the index of the last element. Take care when using this value to iterate over an array not to access the element **ARRAY[@ARRAY]**.

This same technique can be applied to other data types such as strings. For example, consider the following code from **verbs.library**:

```
string LCNumber zero one two three four five six seven eight nine ten
string UCNumber Zero One Two Three Four Five Six Seven Eight Nine Ten

{+number_upper
if arg[0] < 0 : arg[0] > 10
    write arg[0]
else
    write UCNumber[arg[0]]
endif
}

{+number_lower
if arg[0] < 0 : arg[0] > 10
```

```

    write arg[0]
else
    write LCNumber[arg[0]]
endif
}

```

As it is possible to change the value of an **integer** and a **string** after they have been created, there is also a shortcut for creating arrays of variables with all elements being loaded with a default value. This is done by using the keywords **integer_array** or **string_array** followed by the name of the variable and the number of elements to create. For example, the following line of code will create an array of 10 integer variables called **OPTIONS**, all with the value **0**:

```
integer_array OPTIONS 10
```

It is also possible to specify a custom default value after the size of the array like this:

```
integer_array OPTIONS 10 42
```

This will create ten variable with the name **OPTIONS** and the value of **42**.

Constants

The basic syntax for a **constant** statement is:

```
constant ConstantName Value
```

It is possible to define integer and string constants, the type being inferred from the value. If you would like to create a string constant that contains only a number, enclose the number in double quotes: Below is an example of creating constants. This example is taken from the beginning of The Unholy Grail and defines one integer and three string constants that contain the bibliographical information required by the Treaty of Babel:

```

constant game_title      "The Unholy Grail"
constant game_author     "Stuart Allen"
constant game_version    2
constant ifid            "JACL-002"

```

Unlike a variable, the initial value of a constant is not an optional parameter. A value must be specified and this value will remain unchanged for the duration of the game.



Although a variable can be used wherever a constant can be used, if a value is not to change during the game, there are two advantages to using a constant. The first is that the value cannot be changed by accident using a **set** command. The second is that constants are not saved each time the player makes a move so for this reason they also provide a performance increase over the use of variables.

Synonyms

Synonyms are a way of substituting a word in the player's move for another word. They are defined using the keyword **synonym** followed by the word to be substituted and then the word to be put in its place. Care should be taken when defining synonyms, as duplicate **grammar** statements are often the better approach. Consider the following examples:

```
synonym get          take
```

```
synonym grab      take
```

With the above synonyms defined, the command 'get note' would be translated to 'take note' before being parsed. With the below **grammar** statements defined, but no synonyms in place, the command "get note" would be parsed as is, but will still be mapped to the **take** function.

```
grammar take **here    >take
grammar get  **here    >take
grammar grab **here    >take
```

The problem with the synonym approach is that if we were then to define the following **grammar** statement, we would run into trouble:

```
grammar get out >exit
```

With the above **synonym** defined, the command 'get out' would be translated to 'take out' before being parsed, a sentence that the game would not understand.

There are right and wrong times to use both approaches to broadening your game's vocabulary. Just be sure to take care and consider the potential effects of any synonyms you define.

Filters

There are a few filters defined in **verbs.library**, and chances are you will never need to define any of your own. They are defined using the keyword **filter** followed by the word to be filtered from the player's input before it is parsed.

The following are some examples of filter definitions (taken from **verbs.library**):

```
filter the
filter quickly
```

With the above filters defined, if the player typed the command:

```
quickly take the coin from the bag
```

the parser would process:

```
take coin from bag
```



Filters should be defined very sparingly. They are designed to give the illusion of the parser understanding more than it really does. Although at times this can be good, at other times it can be very, very bad.

Grammar Statements

The basic syntax for a **grammar** statement is:

```
grammar MoveSyntax >FunctionName
```

The JACL Author's Guide

The keyword **grammar** defines a move that may be typed by the player and the function that will be executed when this move is made. The *MoveSyntax* section defines the syntax of the move and consists of one or more parameters that may either be a word to be typed verbatim or one of several special tokens. The last parameter of a **grammar** statement is a greater-than symbol directly followed by the core name of the function to be executed if a move of this format is typed by the player.

The table below details the tokens that can be used as part of the *MoveSyntax*:

Token	Description
*here	Indicates that an object in the current location must be supplied at this point in the move.
**here	Indicates that one or more objects in the current location may be supplied at this point in the move.
*held	Indicates that an object held by the player must be supplied at this point in the move.
**held	Indicates that one or more objects held by the player must be supplied at this point in the move.
*present	Indicates that an object either in the current location or held by the player must be specified at this point in the move.
**present	Indicates that one or more objects either in the current location or held by the player must be specified at this point in the move.
*anywhere	Indicates that an object anywhere in the game world must be specified at this point in the move.
**anywhere	Indicates that one or more objects anywhere in the game world must be specified at this point in the move.
*inside	This scope indicator can only be used as the second noun of verbs that have two nouns. It indicates that an object that is a child of the first noun must be specified at this point in the move.
**inside	This scope indicator can only be used as the second noun of verbs that have two nouns. It indicates that multiple objects that are children of the first noun must be specified at this point in the move.
*location	Indicates that a location in the game world must be specified at this point in the move.
\$string	Indicates that an arbitrary text string must occur at this point in the move. If this text string is to contain spaces, the player must enclose the string in double quotes when typing their command.
\$integer	Indicates that an integer must be supplied at this point in the move.

For example, the following are some valid **grammar** statements:

```
grammar take **here           >take
grammar insert **held in *present >insert_in
grammar set *held to $integer   >set_to
grammar type $string on *present >type_on
```

The first statement says that if the player types the word **take**, followed by one or more objects that are in the current location, then the function **take** should be executed. The second states that if the player types the word **insert**, followed by one or more objects that are being held, followed by the word **in**, followed by an object that is either being held or is in the current location, then the function **insert_in** should be executed. The third

states that if the word **set** followed by a single object that is being held, followed by the word **to**, followed by an integer then execute the function **set_to**. Feel free to add extra **grammar** statements that map to library functions into your program. Keeping these extra **grammar** statements within the game-specific part of your code means that you can upgrade the library at a later date without needing to re-enter your additions. The exact way in which these functions are executed is detailed in the section on [Responding to Player's Moves](#).

When using the tokens **\$integer** and **\$string** the values entered by the player will be placed into constants of the same name. If a command contains two or more of the same token an array of constants will be created. For example:

```
grammar send $string to $string    >send_message

{+send_message
write "Sending " $string[0] " to " $string[1] "^."
}
```



When entering a new grammar definition, be sure not to leave a space between the greater-than symbol (>) and the name of the function to be executed.

User Attributes

The basic syntax for an **attribute** statement is:

```
attribute AttributeName [AttributeName AttributeName...]
```

The keyword **attribute** defines a user attribute that can be used throughout your game for any custom purpose you require. Once a user attribute is defined, it is used in the exact same manner as a system attribute. You can define up to 32 user attributes.

Parameters

Parameters are a web-based feature and therefore only be used with the **cgijacl** interpreters. The basic syntax for a **parameter** statement is:

```
parameter ParameterName [Container] [LowestValue HighestValue]
```

A **parameter** is a connection between an HTML form parameter and a JACL container or string. When a defined parameter is passed, its value is copied into the specified container or string before the player's move is processed. A group of two integers may also be specified after the container in the parameter definition. These numbers indicate the parameter's range bounds if it the container is for storing integers. The first integer is the lowest possible value for the parameter, the second is its highest. When a parameter is updated, it will automatically be adjusted to fall within the supplied bounds if required. If no low and high range bounds are specified, they are set to -65535 and 65535 respectively.

Below is a complete JACL program demonstrating the use of **parameter** statement in combination with an HTML form.

```
#!/../bin/cgijacl

parameter TEST  kryten(status)          1          4
```

The JACL Author's Guide

```
parameter PASSWORD password

location thehere : here

integer INDEX
string password

{form
write "STATUS: " kryten(status) ^
write "PASSWORD: " password ^
write "<hr>"
write "<form>"
hidden

set INDEX = 1
repeat
  write "<input type=radio name=TEST value=~" INDEX ~
  if INDEX = kryten(status)
    write " checked" endif
  write ">" INDEX
  set INDEX + 1
  write "^"
until INDEX = 5

write "<p><input type=~password~ name=~PASSWORD~>^"
write "<input type=submit>"

write "</form>^"
}

object kryten : myself
  player

{+intro
execute here.form
}

grammar blankjacl >form
```

When accessed, this program will look as follows in a web browser:

param_test - Mozilla Firefox

File Edit View History Bookmarks Tools Help

http://localhost:8080/?user_id=3455

param_test

STATUS: 3
PASSWORD: secret

☐ 1
☐ 2
☒ 3
☐ 4

Submit Query

Done

Selecting one of the radio buttons and entering text in the password input field then clicking the submit button will cause the value of **kryten(status)** and **password** to be updated before the player's command is processed. As there is no move to process, a **blankjacl** command is proxied on the players behalf. This command has been mapped to the **form** function, which re-displays the form.

CSV Files

A comma-separated values (CSV) file stores tabular data (numbers and text) in plain-text form. Each column in the file is, as the name suggests, separated by commas. JACL supports both the reading and writing of CSV files. By default stores CSV files in a directory named **data** that in the same location as the program that is accessing them. For example, the demonstration program **contacts.jacl** lives in the **projects** directory and stores its data in **projects/data**. When referring to a CSV file using any of the commands detailed in this section only a portion of the filename is specified. This supplied name is used to build the full name of the file using the following formula:

```
<DataDirectory>/<ProgramName>-<Name>.csv
```

For example, the program **contacts.jacl** stores its contacts in a CSV file called **data** which translates to the full file:

```
projects/data/contacts-data.csv
```

This system ensures that no two JACL programs can overwrite each others data or access arbitrary files from the filesystem of the webserver.

The ITERATE and ENDITERATE Commands

Any **ITERATE** command must be of the following format:

```
iterate <FileName> [skip_header]
...
enditerate
```

The **iterate** and **enditerate** commands mark a looping block of code. The block of code is run once for each row in the file specified directly after the **iterate** command. For example, the **+intro** function of the program **contacts.jacl** loops through the file **data** and looks for the last (highest) ID already assigned to a contact. This number is stored so that the next contact created can be given an ID that one higher, and therefore unique:

```
# GET THE LAST ID
iterate data
  if field[0] < last_id
    set last_id = field[0]
  endif
enditerate
```

Each column of the current row being read is loaded into a string array called **field**. In the case of **contacts.jacl**, the ID is stored in the first column, or field.

Below is another example of the **iterate** command, this time it is used to display the entire contents of the file **data**:

```
iterate data skip_header
  set INDEX = 0
  while INDEX < field_count
    write field[INDEX]
    write ", "
    set INDEX + 1
  endwhile
```

```
write "^"
enditerate
```

There are two points of note with this second example. The first is that the optional flag **skip_header** has been specified after the file name. This flag tells the **iterate** command to ignore the first line of the file as it contains column headers, not data. For example, a CSV such as this:

ID	First name	Last name
0	Stuart	Allen
1	Fred	Fargnargle
2	Zaphod	Beeblebrox

The second is the constant **field_count**. This constant contains the number of columns read for the current row. JACL supports a maximum of twenty columns for any given row.

The UPDATE, ENDUPDATE and INSERT Commands

Any **update** command must be of the following format:

```
update <FileName>
...
endupdate
```

Any **insert** command must be of the following format:

```
insert <Field0> [<Field1>...]
```

The **update** is used to re-write a CSV and therefore provide the opportunity to edit the contents. Below is an example of its use from the **+update** function of **contacts.jacl**

```
update data
  if field[0] = $integer
    insert field[0] firstname_data surname_data email_data home_phone_data mobile_phone_data address_data
  else
    insert field[0] field[1] field[2] field[3] field[4] field[5] field[6]
  endif
endupdate
write "Record " $integer " updated.^^"
```

This function uses the **update** command to modify the details for the contact with the ID (**field[0]**) that is equal to **\$integer**. This is achieved by looping through each line of the file (using the **update** command) and then checking if the current line is the line that is to be edited. If so, the new values are written out (using the **insert** command). If not, the existing values are written back to the file (also using the **insert** command.)

The **update** and **insert** commands are also used to delete a record in **contacts.jacl**. This is achieved by inserting all the existing rows from within an **update** loop except the record to be deleted:

```
update data
  if field[0] != $integer
    insert field[0] field[1] field[2] field[3] field[4] field[5] field[6]
  endif
endupdate
write "Record " $integer " deleted.^^"
```

The APPEND Command

Any **append** command must be of the following format:

```
append <FileName> <Field0> [<Field1>...]
```

The **append** command is used to write a single extra row to the end of a CSV file. If the CSV file doesn't already exist it will be created and the appended row written to the new file. Below is an example of its use from the **+add_record** function of **contacts.jacl**:

```
set last_id + 1
append data last_id firstname_data surname_data email_data home_phone_data mobile_phone_data address_data
write firstname_data " " surname_data " added to database.^^"
```


Internals

Constants and Random

When using a command that requires you to provide a numerical value, such as **if** and **set**, the following values may be used.

Constant	Value
scenery	100
heavy	99
true	1
false, nowhere or null	0
status_width	Set to the number of columns the status window is currently displaying.
status_depth	Set to the number of rows the status window is currently displaying.
objects	Set to the number of objects defined in the game.
interpreter	Set to the interpreter type that is currently running the game, either GLK or CGI .
random	Set to a random number between 1 and the current value of the variable max_rand . By default, max_rand is set to 100.
unixtime	Set to the current time, expressed as an integer representing the number of seconds since midnight on the 1st of January 1970.

Internal Commands

Listed here are the only in-game commands that are internal to the interpreter. All other commands to be interpreted during the course of the game must be defined as a **grammar** statement in the game code.

Command	Description
restart	Will cause the JACL interpreter to ask the user if they are sure, restarting the game from the beginning if they answer yes , returning to the game if they answer no .
info	Will display the version information of the JACL interpreter that you are running and the maximum number of items it was compiled to handle. This limit may be increased by changing the MAX_OBJECTS define statement in the file jacl.h then re-compiling the interpreter.
oops [<i>ReplacementWord</i>]	Replaces the most likely incorrect word in the player's last command with the supplied replacement word.
script	Will cause the current game session to be recorded to the file specified at the ensuing prompt.
unscript	Stop a previously started transcript.
again	This command repeats the player's last move.
undo	Will cause the game to revert to the state it was in before the previous command.
quit	This will cause the JACL interpreter to ask the user if they are sure, closing the interpreter if they answer yes , returning to the game if they answer no .

The Menu Library

There are many situations where it is desirable to present the player with a menu of options from which they can select. The **menu.library** contains a set of functions that assist in the display of menus and this chapter details their use.

There are two fundamental types of menu that can be created using **menu.library**. The first displays the menu options then returns the player to the normal game prompt. From there, if they type a command that contains only a single integer it is treated as the selection of one of the menu options. The second type of menu directly prompts the player to select an option using the **asknumber** command and then re-displays the menu options for a further selection. This type of menu can be further customised by setting the variable **MENU_MODAL** to either **true** or **false**. If **MENU_MODAL** is set to **true** the menu can only be exited by selecting an explicit option that breaks out of the menu loop. If **MENU_MODAL** is set to **false** the menu will exit if the player simply presses enter or types **0**.

Both types of menus can be configured to work in one of two modes: proxy or execute. In proxy mode, the string attached to a menu option is treated as a command and is issued as if typed by the player when that option is selected. In execute mode, the string attached to a menu option is treated as the name of a function to execute when that option is selected. The mode of the menu is determined by setting the variable **MENU_MODE** to either **MENU_PROXY** or **MENU_EXECUTE** before displaying the menu.

We will start by demonstrating the creation of a menu that uses the normal game prompt for selecting options. Below is an example function that creates a simple three-option menu:

```
{+display_menu
set MENU_MODE = MENU_PROXY
execute "+menu_clear_options"
execute "+menu_add_option<Quietly open the door<open door"
execute "+menu_add_option<Sneak out the front door<out"
execute "+menu_add_option<Pick up the book<take book"
}
```

The first thing this function does is set the variable **MENU_MODE** to **MENU_PROXY**. This indicates that the final argument passed to the function **+menu_add_option** is a command to be proxied on the player's behalf.

The second line calls the function **+menu_clear_options**. This function simply erases any previous menu options so a new menu can be build from scratch.

The final three lines add options to the menu by calling the function **+menu_add_option**. This function takes two arguments: the text to be displayed for the menu option and either the in-game command to issue or the function to execute if this option is selected.

When this function is executed it will display:

```
1. Quietly open the door
2. Sneak out the front door
3. Pick up the book
```

```
>
```

The JACL Author's Guide

Typing **1**, **2** or **3** from the command prompt will cause the appropriate command to be issued. This menu is considered active until it is erased by calling the function **+menu_clear_options**.

A looping menu is created by calling the function **+menu_prompt** and passing the name of the function that displays the menu options as an argument. For example, the following command will display the same menu as above, only using the **asknumber** command to prompt for a selection until the menu is exited:

```
set MENU_MODAL = false
execute "+menu_prompt<+display_menu"
```

Using this approach, the player will not be able to perform any action other than select an option from the menu. The function **+menu_prompt** calls the function **+menu_clear_options** before calling the supplied function to build the menu. For this reason the function **+menu_clear_options** does not need to be called manually before adding options.

With the variable **MENU_MODAL** set to **false**, the menu can be exited at any time by simply pressing enter or selecting 0. If the variable **MENU_MODAL** was to be set to **true**, only a valid menu option can be selected. If you need to provide a mechanism for the player to exit the menu when **MENU_MODAL** is set to **true**, this is done by adding a menu option that sets the variable **MENU_IN_LOOP** to **false** when it is selected.

Below is an example of a menu displayed by calling **+menu_prompt**:

```
> menu
```

```
MAIN MENU:
```

1. About
2. Instructions
3. Restore saved game
4. Save game in progress
5. Exit menu

```
Type a number between 1 and 5:
```


CSV Files

A comma-separated values (CSV) file stores tabular data (numbers and text) in plain-text form. Each column in the file is, as the name suggests, separated by commas. JACL supports both the reading and writing of CSV files. By default stores CSV files in a directory named **data** that in the same location as the program that is accessing them. For example, the demonstration program **contacts.jacl** lives in the **projects** directory and stores its data in **projects/data**. When referring to a CSV file using any of the commands detailed in this section only a portion of the filename is specified. This supplied name is used to build the full name of the file using the following formula:

```
<DataDirectory>/<ProgramName>-<Name>.csv
```

For example, the program **contacts.jacl** stores its contacts in a CSV file called **data** which translates to the full file:

```
projects/data/contacts-data.csv
```

This system ensures that no two JACL programs can overwrite each others data or access arbitrary files from the filesystem of the webserver.

The ITERATE and ENDITERATE Commands

Any **ITERATE** command must be of the following format:

```
iterate <FileName> [skip_header]
...
enditerate
```

The **iterate** and **enditerate** commands mark a looping block of code. The block of code is run once for each row in the file specified directly after the **iterate** command. For example, the **+intro** function of the program **contacts.jacl** loops through the file **data** and looks for the last (highest) ID already assigned to a contact. This number is stored so that the next contact created can be given an ID that one higher, and therefore unique:

```
# GET THE LAST ID
iterate data
  if field[0] < last_id
    set last_id = field[0]
  endif
enditerate
```

Each column of the current row being read is loaded into a string array called **field**. In the case of **contacts.jacl**, the ID is stored in the first column, or field.

Below is another example of the **iterate** command, this time it is used to display the entire contents of the file **data**:

```
iterate data skip_header
  set INDEX = 0
  while INDEX < field_count
    write field[INDEX]
    write ", "
    set INDEX + 1
  endwhile
```

```
write "^"
enditerate
```

There are two points of note with this second example. The first is that the optional flag **skip_header** has been specified after the file name. This flag tells the **iterate** command to ignore the first line of the file as it contains column headers, not data. For example, a CSV such as this:

ID	First name	Last name
0	Stuart	Allen
1	Fred	Fargnargle
2	Zaphod	Beeblebrox

The second is the constant **field_count**. This constant contains the number of columns read for the current row. JACL supports a maximum of twenty columns for any given row.

The UPDATE, ENDUPDATE and INSERT Commands

Any **update** command must be of the following format:

```
update <FileName>
...
endupdate
```

Any **insert** command must be of the following format:

```
insert <Field0> [<Field1>...]
```

The **update** is used to re-write a CSV and therefore provide the opportunity to edit the contents. Below is an example of its use from the **+update** function of **contacts.jacl**

```
update data
  if field[0] = $integer
    insert field[0] firstname_data surname_data email_data home_phone_data mobile_phone_data address_c
  else
    insert field[0] field[1] field[2] field[3] field[4] field[5] field[6]
  endif
endupdate
write "Record " $integer " updated.^^"
```

This function uses the **update** command to modify the details for the contact with the ID (**field[0]**) that is equal to **\$integer**. This is achieved by looping through each line of the file (using the **update** command) and then checking if the current line is the line that is to be edited. If so, the new values are written out (using the **insert** command). If not, the existing values are written back to the file (also using the **insert** command.)

The **update** and **insert** commands are also used to delete a record in **contacts.jacl**. This is achieved by inserting all the existing rows from within an **update** loop except the record to be deleted:

```
update data
  if field[0] != $integer
    insert field[0] field[1] field[2] field[3] field[4] field[5] field[6]
  endif
endupdate
write "Record " $integer " deleted.^^"
```

The APPEND Command

Any **append** command must be of the following format:

```
append <FileName> <Field0> [<Field1>...]
```

The **append** command is used to write a single extra row to the end of a CSV file. If the CSV file doesn't already exist it will be created and the appended row written to the new file. Below is an example of its use from the **+add_record** function of **contacts.jacl**:

```
set last_id + 1
append data last_id firstname_data surname_data email_data home_phone_data mobile_phone_data address_data
write firstname_data " " surname_data " added to database.^^"
```


Appendix A: JACL Attributes

BIT	DECIMAL VALUE	OBJECT	LOCATION
1	1	CLOSED	VISITED
2	2	LOCKED	DARK
3	4	DEAD	ON_WATER
4	8	IGNITABLE	UNDER_WATER
5	16	WORN	WITHOUT_AIR
6	32	CONCEALING	OUTDOORS
7	64	LUMINOUS	MID_AIR
8	128	WEARABLE	TIGHT_ROPE
9	256	CLOSABLE	POLLUTED
10	512	LOCKABLE	SOLVED
11	1024	ANIMATE	MID_WATER
12	2048	LIQUID	DARKNESS
13	4096	CONTAINER	MAPPED
14	8192	SURFACE	KNOWN
15	16384	PLURAL	
16	32768	FLAMMABLE	
17	65536	BURNING	
18	131072	LOCATION	LOCATION
19	262144	ON	
20	524288	DAMAGED	
21	1048576	FEMALE	
22	2097152	POSSESSIVE	
23	4194304	OUT_OF_REACH	
24	8388608	TOUCHED	
25	16777216	SCORED	SCORED
26	33554432	SITTING	
27	67108864	FIRST	FIRST
28	134217728	DONE	DONE
29	268435456	GAS	
30	536870912	NO_TAB	NO_TAB
31	1073741824	NOT_IMPORTANT	NOT_IMPORTANT
32	2147483648	(Sign BIT)	(SignBIT)

Appendix B: Library Verb Functions

Below is a list of all the functions that are associated with **grammar** statements in **verbs.library**. It is not, however, a complete list of all the possible command constructs the player can use. Although each function is only listed here once, in most cases there are many **grammar** statements mapping to that function. For example, consider the function **+cut**:

```
grammar cut *present >cut
grammar chop *present >cut
grammar stab *present >cut
```

As you can see, the function **cut** has three **grammar** statements that map to it. The list below, however, will only show the first **grammar** statement that maps to the function. To see all the commands that will cause any one of the below functions to be called, search for the function name in the **verbs.library** file.

shake *held	>shake
xyzyz	>xyzyz
verbose	>verbose
brief	>brief
help *present	>help_other
hint one	>first_hint
hint two	>second_hint
hint three	>third_hint
hint four	>fourth_hint
help	>help
help games	>help_games
hug *present	>hug
kiss *present	>kiss
pick *present with *held	>pick_with
rub *present	>rub
rub *held on *present	>rub_on
rub *present with *held	>rub_with
lick *present	>lick
pull *present	>pull
cut *present	>cut
cut *present with *held	>cut_with
break *present	>break
clean *present	>clean
clean *present with *held	>clean_with
yell at *present	>yell_at
pay *present	>pay
order *carried	>order
knock on *present	>knock_on
play *present	>play
take *here with *held	>take_with
fill *held from *present	>fill_from
fill *held with *present	>fill_with
fill *present	>fill
stand	>stand
lie on *present	>lie_on
sit	>sit
sit on *here	>sit_on
drop *held	>drop
take *here	>take
insert *held on *present	>insert_on
insert *held in *present	>insert_in
ask *present for *carried	>ask_for
tell *present about *carried	>tell_about
ask *present about *carried	>ask_about

The JACL Author's Guide

give *held to *present	>give_to
move *present	>move
read *present	>read
look down	>look_down
look up *anywhere in *present	>look_up_in
consult *present about *anywhere	>consult
feel *present	>feel
smell	>smell
smell *present	>sniff
taste *present	>taste
look in *present	>look_in
examine *present	>examine
i	>inventory
inv	>list_inventory
sorry	>sorry
thanks	>thankyou
score	>score
open *present	>open
close *present	>close
lock *present	>lock
lock *present with *held	>lock_with
unlock *present	>unlock
unlock *present with *held	>unlock_with
show *held to *present	>show_to
untie *present	>untie
tie *held	>tie
tie *held to *present	>tie_to
attack *present	>attack
attack *present with *held	>attack_with
wave	>wave
wave to *present	>wave_to
jump on *here	>jump_on
jump over *here	>jump_over
jump	>jump
yes	>yes
no	>no
why	>why?
blow *held at *here	>blow_at
throw *held at *here	>throw_at
remove *present from *present	>remove_from
remove *present	>remove
wear *held	>wear
talk to *present	>talk_to
eat *present	>eat
drink from *present	>drink_from
drink *present	>drink
flick *present	>flick
press *present	>press
light *present with *held	>light_with
light *present	>light
extinguish *present	>extinguish
swim west	>swim_west
west	>west
swim east	>swim_east
east	>east
swim south	>swim_south
south	>south
swim southeast	>swim_southeast
southeast	>southeast
swim southwest	>swim_southwest
southwest	>southwest
swim north	>swim_north

The JACL Author's Guide

north	>north
swim northeast	>swim_northeast
northeast	>northeast
swim northwest	>swim_northwest
northwest	>northwest
climb up *here	>climb_up
climb down *here	>climb_down
enter *here	>enter
climb *here	>climb
swim in	>swim_in
in	>in
swim out	>swim_out
out	>out
swim up	>swim_up
up	>up
swim down	>swim_down
down	>down
look	>look_around
look under *present	>look_under
look behind *present	>look_behind
look through *present	>look_through
look through *held at *present	>look_through_at
look at *present through *held	>look_at_through
wait	>wait
listen	>listen
listen to *present	>listen_to
use *present	>use
turn *present	>turn
turn on *present	>turn_on
turn off *present	>turn_off
pour *present	>pour
tip *present on *present	>pour_on

Appendix C: Tutorial Game Source Code

```
#!/../bin/jacl

constant game_version      1
constant game_title        "Tutorial Game"
constant game_author       "Stuart Allen"
constant ifid              "JACL-512"

string  command_prompt     "^What do you want to do now? "

constant turns_since_last_sip      5

attribute EXAMINED

location bedroom : master bedroom
  west      bathroom
  south     nowhere

{look
print:
  You are in your bedroom. There is a large, soft bed
  in the centre of the room while a doorway to the
  west leads into the bathroom.
.
write "To the south there is "
if door has CLOSED
  write "closed"
else
  write "open"
endif
write " door.^^"
}

{movement
if compass = south
  if destination = nowhere
    write "The bedroom door is closed.^^"
    return true
  endif
endif
return false
}

object bed: bed
  short      a "bed"
  mass       scenery

{look_under
if guide(parent) = limbo
  write "Hidden under the bed you find this week's "
  write "television guide.^^"
  set guide(parent) = here
  points 50
  return
endif
write "You don't find anything else.^^"
}

object guide: television tv tele guide
  short      a "television guide"
```

The JACL Author's Guide

```
long          "The television guide is here."
parent        limbo
mass          5

{examine : read
write "It contains a listing of this week's programmes.^^"
}

{give_to_rick : show_to_rick
print:
    ~Cool!~ Rick exclaims as he snatches it from your
    hands.^^
    Satisfied that you have achieved at least one thing
    today, you decide to go back to bed.^^
.
points 50
execute "+game_over"
}

object door : bedroom door
short        the "bedroom door"
has          CLOSABLE CLOSED

{open_override
set bedroom(south) = living_room
set living_room(north) = bedroom
return false
}

{close_override
set bedroom(south) = nowhere
set living_room(north) = nowhere
return false
}

location bathroom : bathroom
east          bedroom

{look
print:
    You are in the bathroom. The only exit
    from here is back east into the bedroom.^^
.
}

{movement
if compass = east : compass = out
    write "You bang your head as you walk through the "
    write "doorway.^^"
    return false
endif
write "The only exit from here is to the east.^^"
}

object box: small wooden box
has          CONTAINER CLOSABLE CLOSED
short        a "small wooden box"
long          "There is a small wooden box here."
mass          25
quantity      20

{close_override
```

The JACL Author's Guide

```
write "The lid creaks as you push it closed.^"
ensure box has CLOSED
}

object note: orange note
  short      an "orange note"
  long       "An orange note rests on the ground."
  parent     box
  mass       5

{read : examine
write "The note reads, ~Welcome to Jamaica and have a nice day.~^"
}

location living_room: living room
  short      the "living room"
  north      nowhere

{look
if here has VISITED
  write "You have returned to the living room.^"
else
  print:
    You are in the living room. There is a small
    television perched on a low-lying table in front
    of a sofa.

  .
  write "To the north there is "
  if door has CLOSED
    write "closed"
  else
    write "open"
  endif
  write " door.^"
endif
}

{movement
if compass = north
  if destination = nowhere
    write "The bedroom door is closed.^"
    return true
  endif
endif
return false
}

{eachturn : *eachturn_bedroom
move door to here
}

object television: television tv tele
  short      a "television"
  mass       scenery

{examine
if self has EXAMINED
  write "It's Rick who is the TV addict, not you.^"
  return
endif
write "There is currently a cartoon showing on the "
write "television.^"
```

The JACL Author's Guide

```
ensure self has EXAMINED
}

{turn_off
print:
    As you reach over and switch off the television,
    you get quite a shock to see Rick rapidly growing
    a coat of hair and foaming at the mouth. The shock
    of this is only surpassed by that of him sinking
    his newly-acquired fangs into your throat.^
.
execute "+game_over"
}

{turn_on
write "The television is already on."
}

object rick: son boy teenager rick
    has          ANIMATE
    short        name "Rick"
    long         "Rick is here, watching television."
    mass         heavy

{examine
write "Rick is staring blankly at the television screen.^"
}

{tell_about_guide
print:
    ~Oh, you found it, great. Give it to me!~ Rick exclaims.^
.
}

{tell_about_kryten
print:
    Rick starts to nod off as you tell him all about your early childhood.^
.
}

{ask_about_rick
print:
    ~My life is all but ruined until you find the TV guide.^
.
}

{ask_about_television
print:
    ~It could be bigger,~ Rick says with a sigh.^
.
}

{talk_to
print:
    ~Uh, yeah, I'll do it in a minute,~ Rick mumbles
    with out looking up. You have quite a strong
    suspicion that he didn't really hear a word
    you said.^
.
}

#-----
```

The JACL Author's Guide

```
# GLOBAL FUNCTIONS
#-----

{+eachturn
set rick(turns_since_last_sip) + 1
if rick(turns_since_last_sip) = 5
    if here = living_room
        write "Rick takes a sip from his drink.^"
    endif
    set rick(turns_since_last_sip) = 0
endif
}

{+default_ask_about
if noun1 = rick
    print:
        Rick pokes out his bottom lip then blinks several
        times. This, you have figured out over the
        years, translates to, ~Not a clue.~^
    .
    return
endif
return false
}

{+intro
style bold
write "^^" game_title
style normal
write " by " game_author "^^"

print:
    Your alarm rings and you climb out of bed.
    Monday morning again so soon. Oh well, at least
    your house doesn't have a front door so you have
    a good excuse for not going to work.^^
.

if here hasnt OUTDOORS
    set north_wall(parent) = here
    set south_wall(parent) = here
    set east_wall(parent) = here
    set west_wall(parent) = here
endif
set ground(parent) = here
look
}

object kryten: myself self me
has          ANIMATE
short        name "yourself"
quantity     42
parent       bedroom
player

{examine
write "As beautiful as ever.^"
execute "+inventory"
}

grammar about >about
```

The JACL Author's Guide

```
{+about
write "This is the game from the tutorial section of the JACL Author's Guide."
write "See the appendix for the full source to this program.^"
}

integer OFFSET
integer INDEX
string status_text "temp"
constant status_window 3

{+update_status_window
style reverse
padstring status_text " " status_width
write status_text
cursor 0 1
write status_text
cursor 0 2
write status_text
cursor 1 1
write here{The}
setstring status_text "Score: " score " Moves: " total_moves
set OFFSET = status_width
length INDEX status_text
set OFFSET - INDEX
set OFFSET - 1
cursor OFFSET 1
write status_text
style normal
}

object north_wall: north north wall
has NO_TAB
short the "north wall"

object south_wall: south southern wall
has NO_TAB
short the "south wall"

object east_wall: east eastern wall
has NO_TAB
short the "east wall"

object west_wall: west western wall
has NO_TAB
short the "west wall"

object ground: ground floor
has SURFACE NO_TAB
short the "ground"

#debug "debug.library"
#include "webinterface.library"
#include "webinterface.css"
#include "npc.library"
#include "utils.library"
#include "verbs.library"
```


Glossary

association	A function is said to be associated with the nearest object or location that is defined above it in the game file. An underscore and the label of the item it is associated with is appended to the function name specified in the code to form the function's full internal name.
attribute	An attribute is a boolean value, or flag, that can be set on or off (using the ensure command) for any given item.
child	An object is said to be a child of whichever other item its parent element is currently set to. This indicates that the child object is somehow contained within or possessed by the parent item.
command	A command is one of the basic actions that can be performed within a function.
container	A container is any expression that can have its value set to an integer. These are item elements, item pointers and integer variables.
current location	The current location is whichever location the parent element of the object representing the player is set to. The object that represents the current player is specified by the current value of the object pointer player .
directive	A directive is a setting that may be placed in the interpreter's configuration file.
element	An element is any of the properties of an item that can be set to hold an integer value.
expression	An expression is a group of three parameters that may be used with an if command. All expressions evaluate to either true or false .
filter	A filter is any word that has been specified to be removed from the player's move before it is processed by the interpreter.
full internal name	If a function's name begins with a plus sign, it is global and the name specified in the code will also be its full internal name. If not, the name that appears after a function's opening curly brace, followed by an underscore then the label of the item that the function is associated with will be its full internal name.
grammar	A grammar statement is the definition of a command construct that may be used by the player during the game and the function that should be called when that move is made.
grand parent	The grand parent of an object is the last object in the chain of possession. For example, if a coin is inside a purse that is in turn inside a bag, the bag is said to be the grand parent of the coin, while the purse is its parent.
integer	

The JACL Author's Guide

An integer is a 32 bit signed whole number. It can range between -2147483648 and 2147483647.

item	Item is the group term for objects and locations.
keyword	Keywords are any of the terms that define the nine basic data elements in a game. They are object , location , filter , synonym , variable , grammar , constant , string and parameter .
label	A label is the internal name given to an object, location or variable. No two labels can be the same.
location	A location is an item that the player may be inside. All locations have the attribute LOCATION and are defined using the keyword location .
name	All items must have one or more names. These are the names that the player may use to refer to that item during the course of the game.
object	An object is an item that represents any aspect of the game that the player must be able to refer to, but not be inside.
parameter	The term parameter refers to one of two things. Firstly a parameter is any information that follows a JACL command, keyword or property. If multiple parameters are specified, they must be separated by white space.
parent	An object's parent is the item that it is currently inside or possessed by. The parent of the object that represents the player is the location they are currently in.
pointer	A pointer is a special type of variable that may be used anywhere an item label can be. When used, it will be substituted for the label of the item that it is currently set to.
indefinite article	An indefinite article is used before singular nouns that refer to any member of a group. In English these are a or an . An item's indefinite article is defined using the first parameter of a short property. This article will be displayed when using a {list} macro.
definite article	A definite article is used before singular nouns that refer to a particular member of a group. In English this is the . An item's indefinite article defaults to the unless otherwise specified by supplying a definite property. This article will be displayed when using a {the} macro.
property	A property is one of the elements of an object or location that can be defined at start-up.
scope	Scope describes the position of an object relative to the player. The scope indicators are *present , *here , *held and *anywhere .
statement	A statement is a command, keyword or property along with all of its parameters.

The JACL Author's Guide

synonym	A synonym is any word that has been specified to be swapped for another word in all moves typed by the player.
value	A value is an integer, or anything that resolves to an integer such as: an object label, an object element, an object pointer, a variable, true (1), false (0), scenery (100), heavy (99), nowhere (0), lines (the number of lines the screen can display) or random . For more information on constants and random see the chapter on <u>Internals</u> .
white space	White space is any tab, space, comma or colon that occurs outside double quotes

