Chapter 6 – Programming

A Brief Overview of What it Is and How it All Works

Building transcends the VR in that when you `@create` objects and `@dig` rooms you are using the system *as* a computer system rather than acting strictly within the bounds of the MOO's frame story or virtual reality. Programming takes things one step further, in that you create new and original ways for objects to behave. This section presents an abstract overview of how that process works. The following section, "Yib's Pet Rock", is a hands-on tutorial.

First Principles

I take on faith the mechanics of how text travels from your keyboard to the MOO and from the MOO to your screen, and ask you to do so as well. At some later time you may wish to investigate these for yourself, but they are beyond the scope of this book.

The MOO is made of two principal parts, the *server* and the *database*.

The Server

The *server* is the program that runs on the MOO's host machine. It accepts new connections, interprets the *commands* you type, causes various things to happen in the database because of what you type, and causes some text *output* to appear on your screen. A *command* is precisely a line of text that you type with the intention of getting a response from the MOO. This cycle, you typing a command, the server executing it, and output (usually) displaying on your screen is referred to as a *task*, or, more accurately, a *foreground task*.

The Database

The *database* is the entirety of all the *objects* on the MOO, including all their *properties* and *verbs*. (The *core database* is the database that is there when a new MOO first starts, before any new objects, properties, or verbs have been added.) A *property* is a named piece of data associated with a particular object. Where do properties come from? They are added to objects on an as-needed basis by programmers, using the `@property` command. A *verb* is a named sequence of instructions that the server carries out (or executes). On MOOs, the terms *program*, *command*, and *verb* are often used interchangeably.

The Parser

A command consists of one or more words that you type, separated by spaces. One of the things that the server does is analyze (parse) the line of text that you type, and try to identify which verb on which object it should execute. The part of the server that does this is called the *parser*. The first word of the command you type is the name of the verb. The rest the words you type (if any) are called *arguments*, which are items of information that the verb needs in order to work properly. Let's consider a few example commands and talk about their arguments:

```
take rabbit from hat
put hat on table
pet rabbit
page help I'm trying to understand parsing, can anyone
explain it to me?
home
```

For every command, the parser tries to identify an object with a verb of that name on it that it can run, and it considers sets of objects and their associated verbs in a particular sequence. This sequence is the player emself, any feature objects that the player has, the room the player is in, the direct object of the command, and the indirect object. Looking at these examples, you might intuitively figure out that if there is a `hat` nearby, with a verb that lets one `take` things `from` it, that might be an appropriate verb to run, and you would probably be right. Then, if there is a `table` in the vicinity, and a verb that lets one `put` things on it, that might be an appropriate choice, and so on. Some commands, like `page` and `home` don't take direct objects, prepositions, and indirect objects. They take other items of information instead, or no information.

When a programmer creates a verb, e must specify what arguments (if any) the verb uses. These are called *argument specifiers*. When the parser identifies an object and a verb with argument specifiers that are appropriate to the command that was typed in, we say that it *matches* the object or *matches* the verb on the object.

If the parser can't identify an object with an appropriate verb to run, the server sends the following text to your screen:

```
I don't understand that.
```

Tasks

As mentioned above, the cycle of your typing in a command, the parser matching an object and a verb to run, and then output (usually) appearing on your screen is called a *foreground task*. There are three basic kinds of things that any task can do: It can send information (text) to be displayed on your screen. It can modify the database in one or more ways, including changing the values of properties or even creating new verbs to run in future tasks. And *it can start up another task* that does something else, either later or at the same time, but independently. These tasks are called *background tasks*. They can do the same three things that foreground tasks

do, including starting up additional background tasks. Every background task has a unique numerical identification number called its `task_id`. A list of background tasks (identified by `task_id`) that are scheduled to run at a later time is called a *queue*.

How Are Properties and Verbs Created?

There are two commands that are fundamental to the programming process, and these are `@property` and `@verb`. Like any other command, someone types them in (with some arguments), the parser figures out which object is to receive the new property or verb, and then the server runs the verb that causes new properties or verbs to be added to an object.

The `@property` Command

The syntax for adding a new property to an object is:

```
@property <object>.<property name> <initial value>
[<permission flags> [<owner>]]
```

The property name can be anything you want except that it may not contain any spaces. The initial value can be anything you want, but text should be enclosed within double quotes (`""`). The permission flags can be any combination of the letters "`r`", "`w`", and "`c`". They govern who else may access those properties. (A *flag* is a tiny bit of data, usually stored as a `1` or a `0` (often though not always signifying "yes" or "no") within a larger piece of data.) If you include the letter "`r`" in the permission flags, then anyone may read the value of this property, and anyone's verb may access and use the value of this property. If you include the letter "`w`" in the permission flags, then anyone may change the value of the property, and anyone's verb may change the value of the property. The "`w`" flag is hardly ever used; there are safer ways to permit others to vary the value of a property in limited ways that you control. The "`c`" flag controls who is allowed to change the value of the property in the case that someone else makes a child of your object. If you include "`c`" in the permission flags, then the owner of the child object can change it, and your verbs can't change it. If you don't include "`c`" in the permission flags, then your verbs can change the property's value, even on child objects owned by others, but the owners of the child objects can't change the value of the property directly. This is a concept that many people wrestle with, so don't be discouraged if it doesn't make sense right away. It's mentioned several times in the "Yib's Pet Rock" tutorial (page 108), and explained again in the programming reference section (page 156).

When a letter is included as a permission flag, we say that that flag is *set*. When a letter is omitted from the permission flags, we say that that flag is *clear*. For example, the "`r`" flag is usually set, and the "`w`" flag should almost always be clear.

The @verb Command

The syntax for adding a new verb to an object is:

```
@verb <object>:<verb name> <direct object specifier>
<preposition specifier> <indirect object specifier>
[<permission flags> [<owner>]]
```

The verb name can be anything you want except that it may not contain any spaces and should not begin with the asterisk character (*). The argument specifiers are three generalized expressions of the direct object, preposition, and indirect object that are used by the parser when trying to match a verb to run. The direct object specifier and indirect object specifier can be either this or none or any. The preposition specifier may be either any, none, or one of the list of permissible values (such as in or on) given in the Programmer's Reference Manual and the programming reference section (see page 163). The permission flags for a verb can be any combination of the letters "r", "x", or "d". If the "r" flag is set, then others may read the verb. If the "x" flag is set, then other verbs may use this verb as an intermediate step in their own execution. The "d" flag is obsolete but should always be set; it used to govern whether an error, if one was encountered, should cause the verb to cease executing immediately and produce a traceback or be ignored. A later version of the server provided other ways to handle error conditions without causing tracebacks; nevertheless, the programmer's manual indicates that while obsolete, the "d" flag should always be set[17]. A wizard can set the owner of the verb to be someone other than emself.

Here's an example of adding a new verb to an object:

```
@verb collage:paste any onto this rxd
```

(Such a verb might be used to program a collage object so that you could paste anything onto it to create a work of art.)

'Round and 'Round We Go...

After a verb is added to an object, a programmer then sets to programming it, i.e. specifying, in terms the server can understand, just what it is that the verb is to do, either with the @program command (explained in the programming tutorial, page 108) or using the verb editor (explained in the section on using the in-MOO editors). Programming is an "iterative process", which means that it usually takes several tries before a verb works just the way it was originally intended to.

[17] Pavel Curtis, The LambdaMOO Programmer's Manual, section 2.2.3.

The Nitty-Gritty: What Goes On Inside All Those Verbs?

In a nutshell, verbs start up, process data, and then finish.

Starting Up

When you type a command, the first word of what you type is the name of a verb, and you are said to be "invoking that verb from the command line". Sometimes these verbs are called *command-line verbs*. Other verbs, though, are *only* meant to be invoked (or *called*) from within other verbs – they perform some intermediate function and return a result, which the calling verb then uses as if the function had been written into the calling verb itself. These verbs, called from other verbs, are called *subroutines*. Regardless of whether a verb is a command-line verb or a subroutine, all verbs do some initial start up processing when they are called or invoked, and this consists of setting up some *variables*.

A *variable* is like a property in that it is a named piece of data. Unlike a property, however, a variable only exists and has meaning while a verb is running. Also unlike properties, variables aren't stored with objects – so they can't be accessed by other players or other verbs. We say that they are "internal to the verb" or *local*, whereas properties are "external to the verb" or *global*. In the MOO programming language, variables are said to be *dynamically allocated*, which is a fancy way of saying that as soon as a line of MOO-code assigns a value to a variable, voilà! that variable comes into being and contains the value that the verb just assigned to it.

There are different *kinds* of values that variables can hold, and in the computer world, "kinds of values" are referred to as *data types*. In some programming languages, you have to specify at the beginning of a program what variables will be used and what kind of data each will hold. A counter, for example, might be of type *integer*, while a variable intended to hold a person's name would be of type *character string*. In the MOO programming language, you don't have to declare in advance what type of data a variable will hold, and a variable can hold different types of data at different times. There is a way to ascertain what type of data a variable is holding at any particular time, if one needs to know that.

When a verb is first invoked, certain variables are automatically created right away, and are assigned values before anything else happens. These are called *built-in variables*. The data these variables hold are always available for use within the body of the verb itself. They include the object number of the player who typed the command, the direct object (if any), the indirect object (if any), and a special variable called `args`, which holds a list of any other pieces of information the verb or subroutine needs to do its work – in other words, the arguments. For a command line verb, the value of the variable `args` is a list of just those things the player typed – the direct object, the preposition, and the indirect object, or the content of a paged message, for example. Subroutines may need other pieces of information, however. If a subroutine's job is to take a list of numbers and sort them, for example, then it needs to be told what numbers to sort, and that's what would be in its `args` variable.

It is the job of the calling verb to send the right arguments to a subroutine so that the subroutine can do its job correctly.

Processing Data – The Very Stuff

The basic things that verbs do are:

- Change (directly or indirectly) the values of properties on objects in the database.
- Send information to be displayed on someone's screen (or several people's screens).
- Calculate intermediate results from given information and store them in variables. (The "given information" is received by the verb in the built-in variable `args`, which is sometimes also called the *argument list*.)

How is all this done? The server evaluates a sequence of *expressions*. An *expression* is a combination of letters, numbers, punctuation marks and white space which, when evaluated, generates a value. The value of an expression can then be assigned to a variable, or stored in a property, or ignored. Why would a value be ignored? Some expressions have *side effects*, which are actions that occur as a result of evaluating the expression. An example of this would be displaying some text on a player's screen. If all you care about is an expression's side effect(s), then you don't need to store or otherwise pay attention to its value, even though it has one.

The Finish

Calls to verbs are themselves expressions. When all the expressions within a verb have been evaluated, then the verb is said to *terminate*. Any variables that the verb used are removed from the computer's memory, and a value, the final value of the verb, is *returned*, either to the command line or to the verb that called it. If the verb was called from the command line, its return value is ignored. If the verb was called as a subroutine, then its return value may be ignored, or it may be used as a component of a more complex expression.

In Conclusion

The substance of any programmer's manual or programming language reference is an enumeration of the kinds of expressions that are available, what each one does, and (depending on how detailed the reference is) a synopsis of how to use them. Looking at a programming reference can seem daunting, at first, but it isn't an all-or-nothing proposition. If you know a few simple kinds of expressions, then you can write a few simple programs. If you know a wide variety of expressions, then you can write a wide variety of programs, and everything in between. Virtuoso programmers amass a knowledge of expressions and available subroutines the way master chefs

amass a knowledge of ingredients. Anyone who can read can cook, but the more you know, the more you can do.