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.

Yib's Pet Rock: A Programming Tutorial for Beginners

Introduction

This tutorial was originally written on and for LambdaMOO. The overall objective is to give you a footing in the MOO programming environment and a feel for what the programming process is really like. I will take you through the steps of creating and programming a few objects to a fair degree of sophistication. Without the interaction afforded by a classroom setting (or a MOO!), it's all but impossible to avoid your doing at least some of the project by rote, but I hope you'll gain an intuitive sense of at least some of it as you go along. I go light on structured presentations and explain things as and when we need them, and (I hope) just enough so that you can get a handle on the concepts, but not so much that they distract from the project at hand. The point is not to teach each and every nuance in painstaking detail, but rather to give you some exposure to the *kinds* of things that can be done and how to begin doing them.

The MOO Programming Reference section that begins on page 156 is a more structured presentation, and those who prefer to begin with an overview should skip ahead and come back to this chapter afterwards. Both styles of presentation work in concert. A few things mentioned earlier in the book are repeated here, for review and for the benefit of those who may have skipped ahead to this chapter.

I will explain how to inspect the code on an object (yours or someone else's), because this is one of the major methods of expanding one's existing knowledge and horizons. I will suggest a polite way to ask for help from more experienced players.

Nobody writes bug-free code, including this author. If you write your code thinking that it will work perfectly the first (or second) time, you are setting yourself up for disappointment. Debugging is an adventure; it's detective work. If you find a bug, celebrate! You're that much closer to fixing it and moving on to the next one.

I hope to demonstrate what I consider to be good programming style, and to point out ways to improve the inherent quality of your objects by making them robust. It's one thing to whip up a prototype that works, and quite another thing to make a finely-crafted object that is easy for others to understand and use. Although we will be making fairly simple objects, what you learn in their making will transfer to making larger, more complex objects. Polishing is an important part of that process.

Last, by showing you a variety of tools and how to use them, I hope to launch you on a journey that is as much fun for you as mine has been for me.

Preamble and Prerequisites

If you don't already have one, you will need a programmer bit. There are several ways to get one, though in general you have to get one from a wizard. On some MOOs, the process of getting a programmer bit is automated. One common way is to give yourself a gender and description, then send mail to `*wizards` containing the text, “May I please have a programmer bit?”

While this tutorial *can* be done without a copy of the Programmer’s Manual, you’ll get much more out of it if you have one. The programmer’s manual is ordinarily available by FTP from ftp.lambda.moo.mud.org. The file names are:

```
pub/MOO/ProgrammersManual.txt
pub/MOO/ProgrammersManual.ps.Z.
```

The `.txt` one is readable as plain text. The `.ps` one is for printing on a PostScript® printer. The `.Z` means the file has been compressed (use `uncompress` to decode it). (How to use FTP is beyond the scope of this book.)

You will need to know how to edit things on the MOO, which is addressed in Chapter 4 – Using the Mail System and the Editors.

If you find this tutorial too laborious, and/or if you want to go on to do another supervised project after those presented here, see also yduJ’s Wind-up Duck tutorial, which is found in the Library on LambdaMOO.

Some Conventions

Typically, though not universally, commands that call attention to the fact that we’re working on a computer begin with an “`@`” sign. `@edit` is one, `@who` is another. Commands that are consistent with the Virtual Reality usually don’t begin with an “`@`” sign, such as `look` or `drop`.

Angle brackets “`< >`” indicate a place-holder for an actual value that you must supply at the time. For example, if I instruct you to type `examine <object>`, you will type `examine rock` or `examine tree` or `examine book` depending on the actual object of interest. If you are holding an object or if an object is in the same room with you, then you can refer to the object by name. If you aren’t holding or with an object, you can refer to it remotely by using its object number instead. One way to find out an object’s number is to examine it. You can always get a list of objects you own, and their numbers, by typing `@audit me`. You can also type `@audit <player>` to see a list of objects that someone else owns.

It’s nice when objects have help text, and the ones we make will. But if an object doesn’t have help text, the system will tell you to try `examine <object>`. Some people prefer `@examine <object>` instead. What’s the difference? `@examine <object>` will show you *all* the verbs associated with an object, whether they’re meant for you or not. Some people prefer this completeness. `Examine <object>` can be tailored by the object’s owner so that only relevant verbs appear, or so that all the verbs appear in a more logical order, and it’s a more polished look. I will teach you how to tailor what someone sees when e examines an object you have programmed.

Asterisks in verb names: If you type `examine $thing`, you will see:

```
generic thing (aka #5 and generic thing)
Owned by Haakon.
(No description set.)
Obvious verbs:
g*et/t*ake $thing
d*rop/th*row $thing
gi*ve/ha*nd $thing to <anything>
```

The asterisks indicate ways in which you can abbreviate a verb when typing it. So `get $thing` and `g $thing` do the same thing. In a room, you can type `look` or `l` to the same effect. You may put asterisks in your verb names or leave them out. Does it matter? Sometimes. If you're going to embellish an existing verb, then you have to type in the verb name as given. More on this later.

A quick note about formatting. The MOO programming language will let you separate its different elements with any combination of spaces, tabs, and line breaks except that a quoted string, such as `"You have to be holding that to use it,"` may not have a line break in it. (Any line breaks within quoted strings in this tutorial are artifacts of typography.) When typing in commands or lines of MOO code, type in quoted strings as one long line, even if they are longer than a physical line on your screen.

Let's Go!

Well! Let's stop beating around the bush, and make something already! We're going to make a pet rock. Pet rocks don't do much, but it's good to start small. Think of it as a stepping stone to bigger and better things. Type in the following (as one long line):

```
@create $thing named "<your name>'s pet rock","pet
rock","rock","pr"
```

Whew. Let's look at that. `@create` is the command for making a new object What about that "$" sign in `$thing`? Some objects are so basic to the system that they have sort-of universal names, and we never have to remember their object numbers.[18]

When you created the rock, you gave it a list of names, in double quotes, separated by commas. The first one is its actual name. The others are aliases, and you can use any of them to refer to your rock if you are holding it or in the same room with it. If you want to change or adjust the name or aliases, see `help @rename`, `help @addalias` and/or `help @rmalias`. See also `help @create` for all the low-down nitty-gritty on creating things. If you wanted to, you could give it only a name, and no aliases, or you could name it Malcolm, or Fred, or whatever you like.

[18] Where do they come from? They're designated by wizards on object #0. If you were to look at #0.thing you would see #5 (generic thing). So `$thing` means `#0.thing`, and `$note` means `#0.note`, and so on.

Programming is about making choices, and it's the choices you make that make your programming yours.

You also don't have to put quotation marks around the aliases. (I do it out of habit – the two are equivalent.) Speaking of quotation marks, if you want to include the quotation mark character in a string, precede it with the backslash character "\". Names and aliases can't contain commas.

Well, at the moment, our rock doesn't look like much. Let's give it a description. I did mine this way:

```
@describe rock as "A small rock.  It looks friendly, but
doesn't do much."
```

Okay, let's cut to the chase! Time to put a verb on that rock:

```
@verb rock:pet this none none rxd
```

The `@verb` command is how we add commands to objects. Some are "obvious verbs" that show up in `examine`, and others aren't. If you examine your rock now, you won't see the verb you just added. And if you type `pet rock`, the system will respond with `I don't understand that.` (Try it yourself, just to be sure.) That's because we haven't programmed the verb, yet. But hang on, here we go.

There are two ways to accomplish this. One is to type in the verb all at one go, as follows:

```
@program rock:pet
player:tell("You pet the rock.  Nothing happens.");
.
```

Notice the period on a line by itself at the end. That's an essential part.

The other way to program the verb is to use the verb editor, like this:

```
@edit rock:pet
enter
player:tell("You pet the rock.  Nothing happens.");
.
compile
done
```

You should type in each line exactly as given. The semicolons at the end of some lines are part of the program, for example, and the word `enter` is an editor command.

If you get a compiler error message, don't panic. You probably made a typing mistake. See which line it had trouble with (there aren't a lot of choices, with this verb, but there will be, later), and check for a missing semi-colon, mismatched quote-marks, a period instead of a colon, etc. *How* do you check? With compiler errors, you have to do it by using verb editor as shown in the second example above. Type in your verb. Type `compile`. When you get the compiler error, type `list` or `list 1-$` to see the lines listed with their associated numbers. Lines are delimited by semicolons; if you leave one out, the compiler may report the problem as being on the prior or subsequent line than the one the error is actually on. So for an error on a given line, also check the lines before and after the one that the compiler claims is problematic.

Now we're rolling! Examine the rock. Pet the rock. Celebrate!

Got a traceback? Don't panic. Read it, see which line is problematic, then check that line and any lines near it. Tracebacks are *good* (in an intermediate sort of way). They help us find and fix bugs faster.

To fix a program (or change it), you can either type in the corrected version from scratch (`@program rock:pet`) or use the verb editor (`@edit rock:pet`). Note that once you have created a particular verb with the `@verb` command, don't use `@verb` to add it again. That's a once-only operation. Just change it with `@program` or `@edit`.

But What Did I Just Do?

Before we move on, I want to explain a bit more each of the lines I just had you type in.

```
    @verb rock:pet this none none rxd
```

`@Verb` tells the system to add a verb to an object. Let's look at `rock:pet`. "`Rock`" identifies the object. "`:`" signifies a verb, as opposed to a property. "`Pet`" is the name of the verb. The words "`this none none`" are the *argument specifiers* for the verb, and express in a formal way how the command will be typed in by the user. In this case, we want the command to be `pet rock` with no preposition or indirect object. "`rxd`" means that it's readable by others (`r`), callable by other verbs (`x`) (think "eXecutable"), and will generate a nice, informative traceback (`d`) (think "debug") if something goes wrong.

```
    player:tell("Nothing happens.");
```

`Player` is a built-in variable that always refers to the player who typed in the command. `:tell` is one way to cause text to be displayed on someone's screen. The parentheses surround *what* we're going to tell that player. In this case, we've typed in a string, delimited by double quotes, that the player will see. Strings delimited with quotes, object numbers, and some arithmetic numbers are called *literals* (as opposed to *variables*). It is bad programming hygiene to put literals in your verbs (with a few exceptions), and we'll be cleaning that literal out and replacing it with something better shortly.

In fact, let's do that now. Instead of embedding the text in the verb (bad), we'll extract it into a property (good). And this is going to be a special kind of property called a message. Type this:

```
    @property rock.pet_msg "Nothing happens." rc
```

Notice the period instead of the colon between the word "rock" and the string "pet_msg". This, along with the fact that we're using the `@property` command tells the system that we're adding a property and not a verb. `"Nothing happens."` is the *initial value* of our property. We can change it later if we want to. "`r`" means that the property is readable. In general, specify "`c`" when you expect to change this property from the command line, and *not* from within a verb. Leave the "`c`" out if you do intend to change the property from within a verb. (A detailed discussion of what the "`c`" flag means and how to use it begins on page 165.)

Now, edit the verb to use this message. Either `@edit rock:pet` to use the verb editor, or `@program rock:pet` as follows:

```
@program rock:pet
player:tell(this.pet_msg);
.
```

Notice that instead of a string in quotation marks, we've put in a property name instead. "`this`" refers to the object on which the verb is defined, in this case, your rock. "`.pet_msg`" refers to the property we just created.

Now pet the rock:

```
pet rock
Nothing happens.
```

Hmm. Something's missing. Right! I forgot to put in "You pet the rock," first. Well, it's easier to change a property, especially a message property, than to re-edit the verb, and that's one of the reasons why we put messages in properties. Observe:

```
@set rock.pet_msg to "You pet the rock.  Nothing happens."
```

or:

```
@pet rock is "You pet the rock.  Nothing happens."
```

The second form works *because* the property's name ends in `_msg`.

Now pet the rock again. Voilà! See how easy this is? If you wanted to you could try:

```
@pet rock is You pet the rock.  Nothing happens.  What
foolishness!
```

Sharing the Experience

Now let's go public, so to speak, and make our petting action visible to others.

The business of programming has many phases. Among them I number deciding what you want the object to do, thinking up various ways one might do it, seeing if it can be done at all (prototyping), finding a better way to do it. The last one tends to be repeated, and deciding when one has gone far enough is part of what makes it an art.

So. What do I want it to do? I want it to announce to other players in the room that I'm petting the rock.

What are some ways that I might do it? New programmers may not have a clue where to begin. Experienced programmers may have a sense of “the usual way to do it”, if it’s a commonly-done thing, or will investigate how other people have done it, if it’s a thing they’ve seen before. Geniuses may come up with a brilliant, new way to do it that may or may not be practical. For now, follow my lead. Later, I’ll show you some ways to see how other people do things. Good programmers stand on the shoulders of giants (giving credit where credit is due, of course).

We’ll do a prototype, first, to demonstrate that it can be done at all, then I’ll show you some better ways to do it. Here we go:

```
@program rock:pet
player:tell(this.pet_msg);
player.location:announce(player.name +
  " pets the rock.  Nothing happens.");
.
```

`Player.location` is the room where the player is located. `Player.location:announce(<text>)` is a verb defined on all rooms, and it announces text to everybody in the room except `player`, the person who typed in the command. The parentheses contain the arguments to `player.location:announce`. Arguments are the incoming information a verb works with. `Player.name` is the `.name` property of the person typing in the command. Text in double quotation marks is plain text, also called a *literal*. The “+” sign joins, or *concatenates*, two strings of text to each other.

Now to test it. An interesting challenge, here, because you see the regular stuff, but want to know what other people see. You will either need to get a partner, or, if you have the capacity to MOO with two windows at once, log in as a guest, join yourself, and do two things at once. If you work with a partner, I recommend *both* of the following: Pet the rock, and ask your partner what e sees. Ask your partner to pet the rock and see for yourself.

Okay, that’s the prototype. It works, but leaves much to be desired. The first step in improving the situation would be to extract the text into a second message. But wait! The message changes depending on who is petting the rock, so we can only put the fixed part of the message into the property:

```
@property rock.opet_msg " pets the rock.  Nothing happens."
rc
```

This is the part of the message which won’t change.

It is a programming convention on the MOO to make messages in pairs, one for the player doing the action and one for everybody else, and, by convention, the message have the same name except that the one for everybody else is prefixed with an “`o`”, for “others”. To use the message:

```
@program rock:pet
player:tell(this.pet_msg);
player.location:announce(player.name + this.opet_msg);
.
```

Test it. It works, but is only a modest improvement. Modest improvements are progress, though. (If it doesn't work, debug it until it does. Probably a typographical error – we all make them.)

Just as we extracted message text into a property, before, now we're going to extract the business of constructing a message into a separate verb of its own. This may seem like a lot of work for one little message, but the fact is that complex objects have many messages, and we'll be able to generalize our work. So think of it as an investment of effort that will pay off later. (And it will, I promise.)

```
@verb rock:opet_msg this none this rxd

@program rock:opet_msg
return player.name + this.opet_msg;
.

@program rock:pet
player:tell(this.pet_msg);
player.location:announce(this:opet_msg());
.
```

Be sure to test your work.

What's new, here? What's new is that we have just created a verb with the same name as a property. This is fine, and in fact desirable, as I will demonstrate later. For now, just remember to pay close attention to the difference between "`.`" and "`:`".

Also, the new verb had argument specifiers of `this none this`. Because there is no natural English language construct in which a thing is both the direct and indirect object of a verb with no preposition in between, we use this combination of argument specifiers to designate an internal verb (called a *subroutine*) as opposed to a command-line verb. If you examine your rock again, you will note the absence of opet_msg from the list of obvious verbs, which is the way we want it. The new verb returns a result, which in turn is used by the verb that called it.

Plunging Into Pronoun Substitutions!

The time has come to learn to put pronouns into your messages.

At your leisure, you should skim `help pronouns` and `help $string_utils:pronoun_sub`. You don't have to understand every nuance now, but you should know that these help texts exist, and you should have a sense of their scope, for future reference. (I still refer to them from time to time, myself.)

Suppose that when I pet the rock, you wanted the system to announce, "Yib pets the rock. Nothing happens. Doesn't she look foolish!" And when Klaatu pets the rock, we would want it to say, "Klaatu pets the rock. Nothing happens. Doesn't he look foolish!" and when Bits (who use the plural gender) pet the rock, we'd like it to say, "Bits pet the rock. Nothing happens. Don't they look foolish!" (I hope you're beginning to detect a pattern, here.)

Here we go.

The tool that does the work for us is called `$string_utils:pronoun_sub`. It takes a string as an argument, and replaces certain special symbols with values that are meaningful at the time the verb is called. We'll do this in steps. Type:

```
@set rock.opet_msg to "%N pets the rock.  Nothing happens."
```

or:

```
@opet rock is %N pets the rock.  Nothing happens.
```

or:

```
@opet rock is "%N pets the rock.  Nothing happens."
```

Because the property name ends in "_msg" we can set it using any of these forms. (From now on I'm going to let that go without saying.) `%N` is the special symbol for which the player's name will be substituted. (In the "Learn Something New Every Day Department", after all these years of programming, I just noticed that that omitting the double quotes in the second form reduces the number of spaces between the two sentences from two to one, while using the double quotes preserves the two spaces between the two sentences. Fancy that! Since I'm picky about spacing, I'll use double quotes, as I always have (until I started writing this tutorial).)

Now we'll revise the `:opet_msg` verb:

```
@program rock:opet_msg
return $string_utils:pronoun_sub(this.opet_msg);
.
```

Now that the code is in place, lets fancy up that opet_msg a bit more:

```
@opet rock is "%N pets the rock.  Nothing happens.  Doesn't
%s look foolish?"
```

`%s` is the special symbol that substitutes the subject pronoun (he, she, e, they, etc.) Capitalization of the substitution symbols works as you might expect, by the way. Find a way to test this new message with different genders. Either find people of different genders to play with, or find an observer while you set your own gender to different things. Or, if your machine and/or client software has the capability, you can log on simultaneously as yourself and a guest for purposes of testing. Many programmers do this.

Did you find the problem with the plural gender? "Bits pets the rock. Doesn't they look foolish?" Who looks foolish now? The pronouns are good, but there's a little problem with verb agreement. Never fear, `$string_utils:pronoun_sub` will save the day again:

```
@opet rock is "%N %<pets> the rock.  %<Doesn't> %s look
foolish?"
```

Now this works for everybody and everything. (Ain't life grand?) The percent sign in combination with the angle brackets signals to `$string_utils:pronoun_sub` that some adjustment may be needed for verb agreement. From your end, just put the appropriate (English language) verbs between the angle brackets, preceded by the “%” sign. Write them as if they were for a third person singular actor.

Generalizing the Message Verb

What we’ve done so far is work our way up to the .opet_msg property and a corresponding :opet_msg verb that does spiffy pronoun substitutions, and we’ve come quite a long way from where we started out. And we could do another verb, if we wanted, :feed, perhaps, with corresponding .feed_msg and .ofeed_msg properties and an :ofeed_msg verb. (Think about how you might do that.) The :ofeed_msg verb would look an awful lot like the :opet_msg verb, wouldn’t it? In fact, it would bear a STRIKING RESEMBLANCE, except for the name of the message property it referred to. Well, well. Can we capitalize on this and do something more efficient? You bet we can:

```
@program rock:opet_msg
return $string_utils:pronoun_sub(this.(verb));
.
```

Instead of this.opet_msg, I wrote, this.(verb). Notice that the name of the verb is the same as the message. The variable verb is a system-provided or *built-in* variable that contains a string, the name of the verb as it was called. Now do this:

```
@addalias "pet_msg" to rock:opet_msg
```

Just as objects can have aliases, so, too, can verbs. But the important part is that when the verb gets executed, the message on which pronoun substitution is performed and which is ultimately displayed is *either* this.pet_msg or this.opet_msg, *depending on which alias was used to call verb.*

And then:

```
@program rock:pet
player:tell(this:pet_msg());
player.location:announce(this:opet_msg());
.
```

Ta-dah! We’ve now generalized it about as much as we can. Watch closely. Nothing up my sleeve, and presto!

```
@addalias "feed_msg" to rock:pet_msg
@addalias "ofeed_msg" to rock:pet_msg

@prop rock.feed_msg "You try to feed the rock.  Nothing
happens." rc

@prop rock.ofeed_msg "%N %<tries> to feed the rock.  Nothing
happens.  Natch!" rc

@verb rock:feed this none none rxd
```

```
@program rock:feed
player:tell(this:feed_msg());
player.location:announce(this:ofeed_msg());
.

examine rock
pet rock
feed rock
```

Heh.

Polishing the Rock

I am now going to address myself to issues of documentation. At the beginning, writing documentation may seem tedious, and it may seem silly to add comments to such simple programs. But comments and documentation are among the things that separate the good craftsman from the mere hack; adding them consistently is a very good habit to get into.

We'll start with plain help text, which turns out to be easier than you might think. Just create and edit a .help_msg property. (It can have one line or a list of several lines.) I usually start out with an empty list, then edit that with the note editor:

```
@property rock.help_msg {} rc

@edit rock.help_msg
enter
Rocks make great pets!  They're quiet, clean, and easy to
maintain.

Build one today!
.
save
done
```

If you wanted to, you could `@addalias "help_msg" to rock:pet_msg` and do pronoun substitutions in the help messages. Let's do that, just to see:

```
@addalias "help_msg" to rock:pet_msg

@edit rock.help_msg
list
ins 2
enter

This one's name is %t.
```

```
.
save
done

help rock
```

Recall that I asked you to read `help $string_utils:pronoun_sub`. `$string_utils` is an object, and has its own `.help_msg` property. And there is *also* help text for the *verb*, `:pronoun_sub`. How do they do that? If you typed `@list $string_utils:pronoun_sub` (it's rather long), you would see some lines at the top in double quotes, with semicolons at the end. You would probably recognize these as comments, and you would be right. Comments that appear at the top of a verb (before any other lines of code) will also appear as help text for that particular verb. Even though our verbs are small and simple, let's add comments to them:

```
@edit rock:pet
ins 1
enter
"Usage:  pet <this>";
.
compile
done

help rock:pet
```

Do the same for the `:feed` verb. At the top of the `:pet_msg` verb, put a comment that says, "This verb does pronoun substitutions on various messages." Test your work.

And now, for the icing on the cake, because we can, we're going to customize the output when someone types `examine rock`:

```
@prop rock.obvious_verbs {} rc

@edit rock.obvious_verbs
enter
  pet %<what>
  feed %<what>
  give/hand %<what> to <anyone>
  get/take %<what>
  drop/throw %<what>
  help %<what>
 .
save
done


@verb rock:examine_verbs tnt rxd

@program rock:examine_verbs
```

```
"Returns a list of obvious verbs, substituting the string
the player typed in for %<what>";
what = dobjstr;
vrbs = {};
for vrb in (this.obvious_verbs)
  vrbs = {@vrbs, $string_utils:substitute(vrb,
    {{"%<what>", what}})};
endfor
return {"Obvious verbs:", @vrbs};
.

examine rock
examine pet rock
```

The `.obvious_verbs` property should seem fairly obvious to you by now. Let's take a quick look at that `:examine_verbs` verb. It's an internal verb that's called when you `examine` something. It has a comment at the top. `dobjstr` (think "direct object string") is the string the user typed as the direct object in a command. If the player typed `examine rock` then `dobjstr` will be "`rock`". If the player typed `examine pet rock`, then dobjstr will be "`pet rock`" and so on. The next few lines are building a construct to return as a result. We start with an empty list. Then there is a *for loop* that does something with every item of `.obvious_verbs`. What does it do? It does a substitution of `dobjstr`, and appends the new result to the existing list. For the use of the `@` sign in this context, I refer you to `help listappend`. For the rest, parentheses, brackets, and braces nest. It's just a fancy call to a fancy verb, `$string_utils:substitute`, which has its own help text. If you want to, you can take this one at face value, and model subsequent `:examine_verbs` on other objects after this one. Mimicking is a tried and true technique that gets you into hot water sometimes but often works. I'm not above doing it myself when I'm trying to learn how to do something: Mimic something similar, and in the process of getting it to do exactly what I want, I learn how it actually works. Last, notice the return statement: Because of the curly braces `{}`, it's returning a *list* of things, which the calling verb is expecting.

Programming can be wild and woolly, sometimes, but that's part of the fun.

What Can't You Do With a Rock?

Well, you can throw a rock. Throwing rocks isn't very nice. You can, if you wish, prevent people from throwing your rock.

In this case, it's less about programming (more of that later), and more about controlling your environment and the things you own, so let's learn to exercise a bit more of that control.

In MOOs, throwing and dropping are more or less considered synonymous. But it doesn't have to stay that way. `Examine rock`. You will see, among other obvious verbs, `d*rop/th*row rock`. Initial concept: We want dropping the rock to behave normally, but throwing the rock to give the player a message, instead.

Type `@messages rock`. You will see all the `_msg` properties defined on your rock (and all of its object ancestors), including the messages that we added. Notice: No throwing messages. Concept: Make a separate throw verb, and a set of throw messages to go with it. Revision: Actually, they should be no_throw messages, or perhaps, to blend in with what's already there, throw_failed messages. So:

```
@prop rock.throw_failed_msg "Throwing rocks isn't nice, and
besides, this rock likes you, so it stays nestled safely in
your hand." rc

@prop rock.othrow_failed_msg "%N %<makes> a throwing motion
with %t, but can't quite seem to bring %r to let go." rc

@addalias "throw_failed_msg" to rock:pet_msg
@addalias "othrow_failed_msg" to rock:pet_msg
```

So far this should seem familiar. I often start with what I want the messages to be, then verbs to control how, when, and to whom they'll be displayed:

```
@verb rock:throw this none none rxd
```

But wait! The verb name is "th*row", and if we want to override it, we have to name it exactly the same as the verb on its parent. If you went ahead and typed in the line above, remove the verb with:

```
@rmverb rock:throw
```

To be absolutely sure of how to add it, we'll check with this:

```
@display rock:throw
```

We use `@display rock:throw` rather than `examine rock` because who knows how the previous programmer may have gussied up the examine verbs, eh? Based on what we see, then, we'll add the verb like this:

```
@verb rock:th*row this none none rxd

@program rock:throw
"Throwing stones isn't nice.  Thwart that impulse.";
player:tell(this:throw_failed_msg());
player.location:announce(this:othrow_failed_msg());
.
```

Well, on looking at it, that throw_failed_msg is a bit patronizing, and furthermore, unhelpful to someone who actually wants to rid emself of your rock. So let's adjust it:

```
@throw_failed rock is "Throwing rocks isn't nice.  Try
dropping it, instead."
```

Test everything. Try throwing the rock. Try dropping the rock. Examine the rock. Hmm. I found, in my play-testing, that you can't drop a rock if you aren't holding it, but you can throw a rock (or try) if you aren't holding it. We can do better. Let's start by looking at what the original code does:

```
@list rock:drop
```

This will show us the original drop verb. It's pretty old code, and written in an older style. You'll notice the difference between the way that verb handles messages and the way ours do. Our verbs are new and improved. And you'll see some things that you don't understand, perhaps, which may or may not turn out to be relevant. Get what you can out of it and don't worry about the rest, right now. What we're looking for, though, is the part that generates the text, "You don't have that," so that we can do ours in a similar, if not identical way. And what it does is check the location of the rock, and display different messages depending on where it is. Our verb will be simpler, but will behave in a similar way. I'm going to show you two versions, and I hope you can tell by inspection (and maybe by consulting the programmer's manual) what the difference is. They are both equally good, so you can choose which way you want to implement it.

Version 1:

```
@program rock:throw
"Throwing stones isn't nice.  Thwart that impulse.";
if (this.location == player)
  player:tell(this:throw_failed_msg());
  player.location:announce(this:othrow_failed_msg());
else
  "You can't throw a rock if you don't have it in the first
place.";
  player:tell("You don't have that.");
endif
.
```

Version 2:

```
@program rock:throw
"Throwing stones isn't nice.  Thwart that impulse.";
if (this.location != player)
  "You can't throw a rock if you don't have it in the first
place.";
  player:tell("You don't have that.");
else
  "You have it, but you can't throw it....";
  player:tell(this:throw_failed_msg());
  player.location:announce(this:othrow_failed_msg());
endif
.
```

Here is a case where I've chosen *not* to extract a message into a property, so let me tell you why. I put in a perhaps-superfluous comment, "You can't throw a rock if you don't have it in the first place," mostly to set a good example. But in a case where we're just telling the player some sort of error message, the embedded string can serve *as* the comment. So a leaner but just-as-readable version might look like this.

Version 3:

```
@program rock:throw
"Throwing stones isn't nice.  Thwart that impulse.";
if (this.location != player)
  player:tell("You don't have that.");
else
  "You have it, but you can't throw it....";
  player:tell(this:throw_failed_msg());
  player.location:announce(this:othrow_failed_msg());
endif
.
```

Expediency is fine, if it isn't cryptic. If you have to squint to follow what the code is doing, add a comment. You'll be glad later that you did. Trust me.

Before we move on, I want to take a quick moment to point something out in Version 1. Notice the line, `if (this.location == player)`. Notice especially the double-equals sign "`==`". There is a very big difference between a single equals sign and a double equals sign. The first is an assignment statement. `a = b` means, "Set the variable `a` equal to the current value of the variable `b`." `a == b` means, "Is the current value of `a` equal to the current value of `b`?" These are two very different things. Confusing the two is a classic mistake. Heads up.

Here's another way to control your rock and what people do with it.

Suppose you want to lock your rock in place, so that people can't take it. Maybe it's a boulder!

```
drop rock
@lock rock with here
take rock
```

Heh, you can't pick that up, and neither can anybody else. You might change your rock's description to show that it's a boulder. Or you might just change .take_failed_msg and otake_failed_msg to say something like, "It's heavier than it looks, isn't it!"

If you decided to make your rock portable again, type:

```
@unlock rock
```

See also `help @lock` and `help locking`.

Rockin' and Rollin'

So far, we've done a variety of things with our rock, but the rock itself hasn't changed, much. Hardly surprising, I suppose, but the phrase comes to mind, "A rolling stone gathers no moss," and I was thinking, wouldn't it be fun if our rock gathered moss?

How shall we start thinking about this? Well, what if the description had a bit tacked onto the end saying how mossy the rock is? The amount of moss can depend on how long since the rock was last moved.

So. We'll need a list of different amounts of moss. We'll need to write a verb to make the description change over time. We'll need a way to see how long it has been since the rock was last moved, and we'll need a way to convert that amount of time into a phrase about moss.

Let's start with the easy part to get our juices going:

```
@property rock.moss_list {} rc
```

It's going to be a list of text strings, and `{}` is the symbol for the empty list. We start with that. Then:

```
@edit rock.moss_list
enter
It has gathered no moss.
If you were to look at it closely with a magnifying glass,
you would see a tiny bit of moss on it.
There is just a wee bit of moss growing on it.
It has gathered a little bit of moss.
There is some moss growing on it.
It is about half-covered with moss.
It has gathered quite a bit of moss.
It has gathered a great deal of moss.
It is almost completely covered with moss.
It is covered with moss.
.
save
done
```

Time on MOOs is measured in seconds since midnight on 1 January 1970, Greenwich Mean Time. How do I remember that? I don't. It's in `help time()`, which we will be using.

```
@prop rock.last_moved_time 0 r
```

It's `r` (and not `rc`) because we'll be changing it from within our verb (only). Every time the rock moves, we'll record the time. Every time someone looks at the rock, we'll compare the current time to the `.last_moved_time`. Any time an item moves or is moved, its `:moveto` verb is called. We want to take advantage of this by adding a bit to the existing `:moveto` verb, and we do it like this:

```
@verb rock:moveto tnt rxd
```

`tnt` is an abbreviation for `this none this`, our designation for an internal verb.

```
@program rock:moveto
"Reset the reference time (clearing off any moss)";
```

```
this.last_moved_time = time();
"Then do all the usual stuff that the parent does.";
return pass(@args);
.
```

Can we test out this much? You bet. Drop the rock (if you have it) or take the rock (if you don't). You can manually inspect the .last_moved_time property this way:

```
#rock.last_moved_time
```

(See also `help #`.) Move it again, then check `.last_moved_time` again. It changed, by about the number of seconds between moves. Don't be daunted by that great big number. There are plenty of verbs to help us make sense of it. (See `help $time_utils` if you're curious right now.)

We don't care what the number means. It's the *difference* between `time()` and `rock.last_moved_time` – the number of seconds that have elapsed since it was last moved – that interests us. Next we have to pick an interval (measured in seconds) during which a new amount of moss grows. Maybe a day, or a week, but who wants to wait that long to test it? We'll start with, say, a minute, then change the number later after we've tested it:

```
@prop rock.moss_interval 60 rc
```

The next bit is rather a lot of complexity all at once, but try to stay with me, here.

First, let's look at some of those moss descriptions one at a time, to get a feel for them. You'll need to know your rock's object number for this. If you've forgotten it, type `#rock`. Mine is `#70217`, so I'll type that here, but you should use your own rock's object number. (Angle brackets are just too cumbersome for this demonstration.) We're going to play with `eval` a bit. `eval` lets you evaluate a tiny bit of MOO-code on the fly, as it were, without having to write an entire verb to do it. # and `;` are abbreviations for `eval`. It's extremely useful! Try some of these:

```
#70217.moss_list
#70217.moss_list[1]
#70217.moss_list[5]
#70217.moss_list[0]
#70217.moss_list[30]
```

Oho, if we give it too high or too low a number, we get an error. We'll want to keep this in mind when we write our verb. Here are some more examples of `eval`:

```
;length(#70217.moss_list)
;#70217.moss_list[length(#70217.moss_list)]
;#70217.moss_list[$]

;ctime()
;ctime(#70217.last_moved_time)
;time() - #70217.last_moved_time
```

```
;(time() - #70217.last_moved_time) /
   #70217.moss_interval
```

If your moss interval is 60, like mine, the last one will show you how many minutes since the rock was last moved.

Now we're going to add a `:description` *verb* to the rock, which we will then customize. When you look at a thing, the system executes that thing's `:description` verb. For most things, all the verb does is return the value of the `.description` property, very like our message verbs were doing before we got fancy with pronoun substitutions. It's not unusual for objects to have customized `:description` verbs:

```
@verb rock:description tnt rxd

@program rock:description
"Start with the original description, then add to it.";
"Some moss, perhaps.";
base_description = pass(@args);
"What time is it now?";
now = time();
"How long has it been since it was last moved?";
how_long = now - this.last_moved_time;
"How many .moss_intervals is that?";
index = how_long / this.moss_interval;
"If it has been a very short time, index will be 0, but list
elements always start at 1 in MOO code, so we'll add 1.";
index = index + 1;
if (index > length(this.moss_list))
  "It has been so long, index is too high.";
  "So just use the last one.";
  index = length(this.moss_list);
endif
return (base_description + "  " +
  this.moss_list[index]);
.
```

Test your work. Isn't this fun? If you don't like waiting an entire minute each time, set your rock's `.moss_interval` to something smaller, like 10 or 15. Then when you're satisfied, set it to something longer. How many seconds in a day? In a week? In a month? Find out like this:

```
"One hour
;60 * 60
"A day
;60 * 60 * 24
"A week
;60 * 60 * 24 * 7
"And so on.
```

And now, a stationary stone gathers moss.

Looking Under Various (Other) Rocks

Learning a new language is always a challenge, and a programming language is no different. It's nice, as an adult, to attend a language class and have a professor or instructor take you through the material in a logical sequence, so that first you learn to express simple things, then more complex things, until you get a sufficient understanding of the grammar and a sufficiently large vocabulary that you can start to express your own ideas independently. Children learn languages, on the other hand, by being immersed, by imitating those around them, by trying things and seeing which utterances get results and which get perplexed looks from their elders. I don't think anyone learns a language by reading a dictionary.

The Programmer's Manual is a good reference book, but there are other tools that will help you explore the MOO around you and learn (by example) from what's out there already. I am going to detail some of those tools now.

Suppose you want to investigate an object to find out what makes it tick. First, (I hope) you would `examine` the object and perhaps play with it a bit.

Then you might type `help <object>`, to see what the owner or creator wants you to know about it. Then, perhaps, `@parents <object>` to get a handle on what sort of object it is. You'll get a list of object numbers and names, and it may (or may not) be fruitful to check the help text on the object's ancestors, as well.

Then, perhaps, you'd like to know whether this object has any special programming of its own that makes it different from its ancestors. This is where the `@display` verb comes in. It's one of my favorites. There's extensive help text on it, but the form of `@display` that I use most often is `@display <object>.:` which gives a list of verbs and properties defined on that object. (Note that if none are defined, however, you may see the verbs and properties defined on its immediate parent, instead. You can fix that by typing `@display-options +thisonly`.) Often you can get a good start on understanding an object just by looking at the names of the verbs and properties defined on it. Trick: If an object is set as a whole to be unreadable, but some or all of its verbs *are* readable, you can get a handle on those by typing `@verbs <object>` instead of using `@display`. (See also `@show <object>`.)

If a particular verb catches my eye, I might list it, using the command `@list <object>:<verbname>`, just to see what's in there.

Another thing I might do is type `@messages <object>` to get a sense of the scope of its output. If there are messages that I haven't found yet in the course of my playing, then I know that the programming on the object is richer than first meets the eye, and that it might well be worth exploring the object further.

Some people like to use either `@dump <object>` or `@dump <object> with create` to get a complete listing of everything on it. As a rule, I don't care for the large quantity of text that `@dump` generates, but some people like to print out everything there is to know about an object, send it to a printing device, and read the hard copy at leisure. If that's your cup of tea, by all means do that.

Suppose you see a line of text, that you know to be from a particular object, and you want to home in on it to see what verb generates it, and what's going on in the vicinity (if you will) of the line of text that has caught your interest. For example, "Yib tries to feed the rock. Nothing happens. Natch!" The `@grep` command can be helpful here. (Note, `@grep` requires an object number, not the name of an object, even if you are in proximity.)

```
@grep "Natch!" in <object>

Searching for verbs in <object> containing the string
"Natch!" ...

Total: 0 verbs.
```

Well, in this case we struck out. But if you typed `@messages rock` and found that the phrase, "Natch!" was part of the `.ofeed_msg` property, then you could type:

```
@grep "ofeed_msg" in <object>
```

and you would see:

```
Searching for verbs in #<object> containing the string
"ofeed_msg" ...

 <object>:feed [<verb owner>]:
player.location:announce(this:ofeed_msg());

Total: 1 verbs.
```

So in this case you would have learned that `ofeed_msg` (which contains the string we're interested in) is used in the `:feed` verb, so then you might list that verb out to see the larger context.

Last, suppose you are MOOing along, minding your own business (more or less), and out of the blue you see the text, "A black magpie flies in and looks greedily at bright sparkly thing." (You happen to have dropped said sparkly thing recently.) Suppose you made that sparkly thing yourself, and know for sure that there is nothing in its verbs or properties that refers to a black magpie. Where did this line of text come from? What's going on here? You can type `@check-full <text>` and get a trace of the verbs that were called in the process of delivering this choice tidbit of text to your baby blue eyes:

```
@check-full magpie

Traceback for:
A black magpie flies in and looks greedily at bright sparkly thing.
This           Verb               Permissions    VerbLocation   Player
-------------- ------------------ -------------- -------------- ------------
#58337(Y)      tell(1)            #67(Rincewind) #7069(generic) #5720(blue)
#58337(Y)      tell(29)           #3920(Jay)     #33337(PC Clas #5720(blue)
#58337(Y)      tell(4)            #57140(SSO)    #40099(SSSPC)  #5720(blue)
#58337(Y)      tell(1)            #58337(Y)      #58337(Y)      #5720(blue)
#6193(Driveway announce_all(3)    #2(wiz)        #3(generic roo #5720(blue)
#6193(Driveway announce_all(3)    #24442(rw3)    #17755(Integra #5720(blue)
#77522(magpie) make_the_rounds(27 #61050(Y_A)    #77522(magpie) #5720(blue)
#77522(magpie) wake_up(19)        #61050(Y_A)    #77522(magpie) #5720(blue)
-------------- ------------------ -------------- -------------- ------------
```

Well, in this example, you can find out the object number of the magpie, the names of a couple of verbs on the magpie that might be worth looking into, and then you're off and running. Heads up: If you are doing this on LambdaMOO and have the Lag Reduction FO of Godlike Powers, you will have to type `@addlag` and `@paranoid <number>` in order to get anything useful from `@check-full`. `<number>` in the `@paranoid` command refers to a number of lines to keep tracebacks for. There is help text for all of these functions.

Read lots and lots of verbs. Even if you don't understand everything in them, you will gain exposure, start to pick up on patterns, and, as you are ready, absorb new concepts and learn new tricks. Leave no stone unturned.

Asking Others for Help

It is very important that you give a problem the old college try before asking others for help, for a couple of reasons.

First, it's inconsiderate to ask someone else to put more time and work into investigating a bug of yours than you are willing to do yourself.

Second, the very act of trying everything you can think of and checking every reference you know about makes you more receptive to and able to understand the answer when you finally get it.

Before asking for help:

- Read any tracebacks and look at the verb/lines that seem to be causing a problem.
- Read any compiler errors and look at the verb/lines that seem to be causing a problem.
- See if you can find any online help text that addresses your difficulties. (Don't forget `help index`.)
- See if there is any other documentation relevant to your project.

When you ask for help, put together a summary of the problem:

- Include a brief description, and a copy of a traceback (if any).
- Describe in detail how to duplicate the problem.
- Include object numbers -- don't just say, "My pet rock doesn't work."
- Indicate what you've tried so far, both to show that you *have* tried, and to save your helper the trouble of trying things that you've already checked.

DO ask for help if you're genuinely stuck. Most people are happy to assist (or at least try) if you ask nicely and demonstrate respect for their time.

Is It Covered With Moss, Yet? (A Small Side Project)

While my rock was gathering moss, I was wanting to check it each and every .moss_interval, just for the fun of reading all the messages. But I felt silly typing `look rock` over and over again, waiting to see whether it had changed yet. So I thought, "I wish I had a timer, so I could set it and forget it, and be reminded when it was time to look at the rock again."

Let's make one.

Concept: This will not be a complicated object to use. All we need is a verb, `set timer for <some duration>`. One verb ought to do it. We'll get more practice using `$time_utils`:

```
@create $thing named timer
@describe timer as "A simple timer, such as you might find
in the kitchen."
```

Now we'll start with a prototype, to see if we can make it work at all, then refine it and make it more robust. With programming (as with many kinds of projects) the trick is, "Divide and Conquer." If a task seems too big and overwhelming, divide it into subtasks. If those are still too complicated, divide those further. I'm going to show you many versions of one verb, to demonstrate that these programs don't emerge full blown from my head. After years and years of programming, I still work simple-to-complex. Here we go.

First, I typed `@display $time_utils:` (note the colon), which gives me a list of all the verbs on that particular utility package. I have something in mind that I'm looking for, and that just comes with experience and lots of exploring. You could also type `help $time_utils`. Aha, here is what I'm looking for, `$time_utils:parse_english_time_interval`. This will let me take input like, "1 minute" and turn it into a number of seconds. Let's try out just that much:

```
@verb timer:set this for any rxd

@program timer:set
duration =
$time_utils:parse_english_time_interval(iobjstr);
```

```
player:tell(tostr(duration));
.
```

I created the verb, and then put the bare minimum of programming on it. `Iobjstr` is the string that someone types in as the indirect object, and, since it can be anything (it will be a duration, in English words), I gave `this for any` as the arguments. In the program I set up a variable called `duration`, which stores the output from one of the `$time_utils` verbs. My only goal at the moment is to satisfy myself that I can take a string like “10 seconds” and get a number 10 out of it. The second line of the verb is a simple output line, to player (that’s me, the developer) to see if I did, in fact, get something intelligible. Note the `tostr(duration)`. `duration` is a number. `player:tell` takes a string. `tostr()` turns a number into a string. So I’ve turned the number into a string (inner set of parentheses), then I’m telling that string to `player` (me):

```
set timer for 10 seconds
  10
```

Success! But let’s test further:

```
set timer for 10
Incorrect number of arguments
set timer for Fred
Incorrect number of arguments
```

Hmm. I want to know if $time_utils:parse_english_time_interval is sending back the STRING “Incorrect number of arguments”, in which case I can work with it, or whether the system is giving me this message, in which case I’ll have to scratch my head considerably more. I’m going to adjust my verb to answer this question:

```
@program timer:set
duration = $time_utils:parse_english_time_interval(iobjstr);
player:tell("Result: " + tostr(duration));
.
```

All I’ve done is insert the word “Result: ”, before the result I get back from $time_utils. If I’m getting the string back from the utility, then I’ll see the word, “Result: ” at the beginning. If I’m getting the message from the system, then I won’t:

```
set timer for 10
Result: Incorrect number of arguments
set timer for Fred
Result: Incorrect number of arguments
```

Yay! The message is coming from $time_utils, so I’ll be able to snag it easily and deal with it gracefully. (In fact, there are ways to intercept system error messages, too. See the note for the ERR datatype on page 158).

```
@program timer:set
duration =
  $time_utils:parse_english_time_interval(iobjstr);
if (typeof(duration) == NUM)
```

```
    "We got a good value for duration.";
    player:tell(tostr(duration));
  else
    player:tell($string_utils:pronoun_sub(
      "Try something like 'set %t for 3 minutes'."));
  endif
  .
```

This version of the verb tests to see whether the result returned by `$time_utils:parse_english_time_interval` was a number or not. If it's a number, then we got good input. If we didn't get a number back, then we'll give the player a polite and instructive error message. The `typeof()` built-in function is what I use to find out what sort of thing I'm working with. See `help typeof()`. At this stage, I choose not to extract that error message into its own verb, so I do the pronoun substitution on the fly. With more than a handful of lines, it's also time to put help text at the top of the verb, so I'll be sure to do that in the next round. Test the code as before. When you've elicited every message for every contingency you've accounted for, then you've done enough.

Now it's time to make the timer actually do its thing (drum roll, please):

```
  @program timer:set
  "Usage:    set <this> for <duration>";
  "Example:  set timer for 1 hour";
  duration =
    $time_utils:parse_english_time_interval(iobjstr);
  if (typeof(duration) == NUM)
    "We got a good value for duration.";
    fork (duration)
      player:tell("Ding!");
    endfork
    player:tell("The timer starts ticking.");
  else
    player:tell($string_utils:pronoun_sub(
      "Try something like 'set %t for 3 minutes'."));
  endif
  .
```

You should be able to follow most of this. The only new items are those `fork` and `endfork` statements. When you set a timer, you set it down and go off and do something else, while the timer does its thing independently. And that's what's going on here. Everything between the lines, `fork....endfork` will be done at a later time. How much later? The number of seconds that we give to `fork` as an argument, in this case, duration. Anything after the `endfork` statement gets done right away. Try this out, now. Use a duration like `1 minute`. An hour is probably longer than you want to wait.

You can check on background tasks (as these are called) with the `@forked` command. See `help @forked` and also `help @kill`. Forked tasks hog system resources, and should only be used in moderation. If you work on a MOO that has a task scheduler, you should make a point of learning to use it (when you feel ready).

Well. When I tried it, it worked, but the "Ding!" got lost in some other text I was displaying on the screen at the time. So on the next round, I'm going to indent it, among other things. The only thing new is that I'm going to make the output prettier. Pretty output is more important than you might think. What I want is, "Ding! 10 seconds are up", indented so that I'll notice it more. I also want it to differentiate between "60 seconds *are* up", and "1 minute *is* up":

```
@program timer:set
"Usage:    set <this> for <duration>";
"Example:  set timer for 1 hour";
duration = $time_utils:parse_english_time_interval(iobjstr);
if (typeof(duration) == NUM)
  "We got a good value for duration.";
  fork (duration)
    "Indent the message, for better visibility.";
    "Add text to indicate how much time has elapsed.";
    be = ((iobjstr[$] == "s") ? "are" | "is");
    message = iobjstr + " " + be + " up.";
    message = $string_utils:capitalize(message);
    player:tell("  Ding!  " + message);
  endfork
  player:tell("The timer starts ticking.");
else
  player:tell($string_utils:pronoun_sub(
    "Try something like 'set %t for 3 minutes'."));
endif
.
```

The new material is within the fork/endfork statement. I construct a message using smaller strings and the "+" sign to concatenate them together. Here's the mystery line:

```
be = ((iobjstr[$] == "s") ? "are" | "is");
```

Here it is in pseudo-code:

```
if (the last letter of iobjstr is "s")
  set the variable 'be' to the string "are"
else
  set the variable 'be' to the string "is"
endif
```

Divide and conquer. Working from the inner-most parentheses outwards: `Iobjstr` is going to be something like, "10 minutes", or "1 hour". I want to know whether the last character is an "s". `iobjstr[5]` means the fifth letter of iobjstr. `iobjstr(length(iobjstr))` is the last letter of `iobjstr`. `iobjstr[$]` is a way to abbreviate that. "$" in this context means "last". Note the double "`==`" sign. If you are assigning a value to a variable, use one: `x = 3`. If you are testing for equality, use two: `if (x == 3)...` These two different usages lend themselves to typographical errors. Be sure to look for a mistake in the number of "=" signs when you are debugging.

Now what about that question mark? Am I uncertain of what I'm coding? No. The paired symbols, "?" and "|" are an abbreviated way to do a simple `if...then` statement:

```
result = (x ? a | b);
```

is the short way of writing:

```
if (x)
  result = a;
else
  result = b;
endif
```

"`x`" can be an arbitrarily complicated expression.

So:

```
be = ((iobjstr[$] == "s") ? "are" | "is");
```

sets up a variable, `be` to be either the string `"are"` or the string `"is"` depending on whether `iobjstr` ends in the letter `"s"` or not.

If you've played around with your timer, you may have noticed that you can set it more than once. It's actually a multiple timer. We could call this a bug, and add code to see if the timer is ticking, and, if it is, tell the player that it's currently in use. Or we could call this a feature, and add code (and documentation!) to take advantage of the fact. I choose the latter.

Good Times

We have a timer, and it times things. It dings when the time is up, and we can set it more than once -- it's a multiple timer. Being the forgetful sort, now I would like to be able to type in an optional reminder message, so that when the timer dings, I'll know what it was I had set it for.

```
@program timer:set
"Usage:    set <this> for <duration>";
"Example:  set timer for 1 hour";
duration = $time_utils:parse_english_time_interval(iobjstr);
if (typeof(duration) == NUM)
  "We got a good value for duration.";
  "Ask for an optional reminder message.";
  message = $command_utils:read("an optional reminder
message");
  if (!message)
    be = (iobjstr[$] == "s" ? "are" | "is");
    message = iobjstr + " " + be + " up.";
    message = "  Ding!  " +
$string_utils:capitalize(message);
  endif
  fork (duration)
    player:tell(message);
  endfork
  player:tell("The timer starts ticking.");
else
  player:tell($string_utils:pronoun_sub(
    "Try something like 'set %t for 3 minutes'."));
endif
.
```

The new line here is:

```
message = $command_utils:read("an optional reminder
message");
```

Oho! Another utilities package, `$command_utils`. Check out its help text to see what sorts of things are in this box of tools. Don't worry about understanding it all; just get acquainted a little bit, for future reference. We are going to read a line of input from the user (`player`), and store its value in the variable `message`. If the user typed `<enter>` without any text, then we'll just build the message as before.

Notice that I took some of the message building out of the fork...endfork construct and did it all ahead of time. This is a matter of programming style, which is hard to teach. My reason was, put all the message building code in one place, before the fork statement. Then when the forked duration is up, just tell the player the message. In a situation where I wouldn't know what the message should be until after the time had elapsed, it would be appropriate to construct or select message text within the body of the fork/endfork block.

Edit or type in the new version, and try it out. This is not bad, but I missed the `Ding!` if I typed in a message. So I'm going to adjust it so that the output is more to my taste. The next changes aren't really substantive, so I've just noted the changes with comments in the code itself.

```
@program timer:set
"Usage:    set <this> for <duration>";
"Example:  set timer for 1 hour";
duration = $time_utils:parse_english_time_interval(iobjstr);
if (typeof(duration) == NUM)
  "We got a good value for duration.";
  "Ask for an optional reminder message.";
  message = $command_utils:read("an optional reminder
message");
  if (!message)
    be = (iobjstr[$] == "s" ? "are" | "is");
    message = iobjstr + " " + be + " up.";
    message = "  Ding!  " +
$string_utils:capitalize(message);
```

```
  else
    "Add a Ding!  because I can.";
    message = "  Ding!  " + message;
  endif
  fork (duration)
    player:tell(message);
  endfork
  player:tell("The timer starts ticking.");
else
  player:tell($string_utils:pronoun_sub(
    "Try something like 'set %t for 3 minutes'."));
endif
.
```

This version does *almost* exactly what I want. The last step (before doing up the examine verbs and the help text) is to take a step back and look at the code and see if I can make it any better. I notice that I am prepending "Ding!" in two places, and can consolidate that. Now that you understand what more of the code means, some of the comments are superfluous, and I'm going to take them out. That last `else` statement is pretty far from its matching `if` statement, so I'm going to add a comment down there. And I found the Ding! message still not quite prominent enough, so I'm going to use a variant on `player:tell` to put it between blank lines. Here is my final version:

```
@program timer:set
"Usage:    set <this> for <duration>";
"Example:  set timer for 1 hour";
duration = $time_utils:parse_english_time_interval(iobjstr);
if (typeof(duration) == NUM)
  "We got a good value for duration.";
  message = $command_utils:read("an optional reminder
message");
  if (!message)
    be = (iobjstr[$] == "s" ? "are" | "is");
    message = $string_utils:capitalize(iobjstr + " " +
      be + " up.");
  endif
  message = "  Ding!  " + message;
  fork (duration)
    player:tell_lines({"", message, ""});
  endfork
  player:tell("The timer starts ticking.");
else
  "Didn't get a good value for duration.";
  player:tell($string_utils:pronoun_sub(
    "Try something like 'set %t for 3 minutes'."));
endif
.
```

Last but not least, we’ll add help text and examine verbs. We’ve done this before:

```
@prop timer.help_msg {} rc

@edit timer.help_msg
enter
This is a simple timer.  To use it, type 'set timer for
<duration>'.
Here are a few examples:

  set timer for 3 minutes
  set timer for 45 seconds
  set timer for an hour

The timer will prompt you for an optional reminder message.

You can time more than one thing at once.  That is, you
don't have to wait until the first time is up before setting
the timer for another thing.  Reminder messages are
especially helpful when you are timing several things
simultaneously.
.
save
done

@prop timer.obvious_verbs {} rc

@edit timer.obvious_verbs
enter
  set %<what> for <duration>
  give/hand %<what> to <anyone>
  get/take %<what>
  drop/throw %<what>
  help %<what>
.
save
done


@verb timer:examine_verbs tnt rxd

@program timer:examine_verbs
what = dobjstr;
vrbs = {};
for vrb in (this.obvious_verbs)
  vrbs = {@vrbs, $string_utils:substitute(vrb, {{"%<what>",
what}})};
```

```
    endfor
    return {"Obvious verbs:", @vrbs};
    .
```

Voilà, a finished product that does something useful! As you can see, objects and verbs evolve, from gleams in their creators' eyes to simple proof-of-concept prototypes, to fancy versions with whistles and bells, to finished products with nice exam verbs and help text.

What's Big and Red and Eats Rocks?

When I first started planning this tutorial, I wanted to make a small pet. Something that one could interact with, something that could respond in a limited way to things happening around it, something that would seem to take on "a life of its own", which is part of the true magic of programming on a MOO. So for our final project, we're going to make a big red rock eater.

First, the brainstorming. Petting the big red rock eater should generate more interesting results than petting a rock. There should be a command of the form, `feed <anything> to Red`. Hmm. What should happen to things that get fed to Red? Should they disappear? Should we make a special place called "`RedBelly`"? Should it be possible to feed a *player* to Red? That might be an interesting experience, but not all players are gracious about being moved. If it eats, then it would be nice to have a good VR way to get things back or have them re-appear. Should it excrete? Maybe it could go hunting and find some sort of treasure, like a cat finding a mouse. Or maybe it could be like a cartoon character and reach into its own innards and produce a previously-eaten object. I'll have to think about that one. Should it talk? How about if it had variable color spots? I want it to have a gender, so we'll make it a kid of the generic gendered object. If it only eats rocks, then I'll need a way of deciding whether a thing is a rock. Or maybe it eats anything, but is especially fond of rocks. How about a verb to tickle the rock eater? Nah. It wouldn't effect any change in state, and emote works just as well. So let's pass on that one. (I know, the same could be said of the :pet and :feed verbs on the pet rock, but those were instructional exercises.) Maybe it would eat rocks in the vicinity spontaneously. Or maybe not. It should have a repertoire of spontaneous actions, things that it "just does" from time to time. I know what cats do, but will have to think of particulars for rock eaters. Still, the concept is good. Maybe it's like a Tamagotchi™, and if you don't take good enough care of it, it dies! Or maybe if it gets hungry enough, it eats its owner and the owner gets booted! (Fun thoughts, but maybe that's taking things a bit too far.) Maybe, though, its description could change depending on how hungry it is -- like the rock gathering moss. That would be a good compromise.

That's how I typically start a project. I write down everything I can possibly think of, adding to the list over the course of several days (sometimes weeks, or even months). I consider possibilities, no matter how far-out, and pose myself problems. Some things I already know how to do, because I've done the same or similar things before. Others I may have to research.

The next step, then, is to create an object and start fleshing it out, making it more complex as I go along:

```
@create $thing named "a big red rock eater", "big red rock
eater", "red rock eater", "rock eater", "eater", "brre",
"Red"
```

I decided at the last minute to name mine “Red”. Feel free to name yours something else. Naming is actually hard, sometimes. You have to decide what you want people to see when they enter a room (for example): `"You see a big red rock eater here,"` or `"You see Red here."` I may yet change my mind and rename him so that the name is `Red`, and `"big red rock eater"` appears in the description, instead. (The system ignores capitalization – so “red” and “Red” would both refer to our big red rock eater – except that it preserves capitalization when displaying text.)

```
@rename big red rock eater to "Red"
```

or how about:

```
@rename red to "Red (a big red rock eater)", "a big red rock
eater", "red rock eater", "rock eater", "eater", "Red"
```

There are some other options that I may explore later, in particular a `:title` verb, which usually but doesn’t always return an item’s name. I might want it to be `"Red (a big red rock eater)"` sometimes, and sometimes just, `"Red"`. I’ll defer that for now, but it’s an option to keep in mind.

```
@describe <your rock eater> as "<your idea of what a big red
rock eater looks like>"
```

Aha! Food will go where any other LambdaMOO food goes, that is, to its home if it has a `.home` property defined on it, otherwise to its owner’s home. Whew, I’m glad to get that off my mind. (Creative development rarely follows a logical, linear sequence.)

Now that I’ve resolved the food issue to my satisfaction, let me show you a fun (optional) thing you can do with the description, which will further enrich your pronoun substitution skills. My rock eater is red, but he has spots, and I want the spots to be a different color each time someone looks. For variety.

Recall that we wrote a `:description` verb to enhance the pet rock’s description with the addition of some moss. Our strategy here will be similar, with a custom description verb. In that verb, we will determine the color of the rock eater’s spots. But we’ll refer to the color in the description itself. Stay with me, here:

```
@property red.color "blue" r
```

Notice, again, that the property is `r` and not `rc`. That’s because I intend to change it from within a verb, later, and not from the command line.

Start simple, then get fancy. For now, he’ll have blue spots. Then we’ll work on variable-colored spots. In my examples, I’m going to give the rock eater a fairly generic description, so as not to meddle with your own idea of the critter’s physiognomy:

```
@describe red as "You see a big red rock eater with
%[tcolor] spots."

@verb red:description this none this rxd

@program red:description
base_description = pass(@args);
return $string_utils:pronoun_sub(base_description);
.
```

Here's what's going on with the new fancy footwork. Look first at the `.description` property. Recall that the "%" sign is a special symbol used by `$string_utils:pronoun_sub`. The square brackets delimit a unit of text that `$string_utils:pronoun_sub` will work on. The initial "`t`" inside the square brackets refers to `this`, a special variable in MOOcode that means the object on which the current verb is defined – here, the big red rock eater. The rest of what's in the square brackets is the name of a property on the object, in this case, `color`, which we just added. So in essence we're telling `$string_utils:pronoun_sub` to substitute `this.color` for `%[tcolor]`, and it does. Take a look at your rock eater now!

If my rock eater were always going to be red with blue spots, I would have just written that into the description and not gone to all the trouble. But I'm laying a foundation.

Next, I want the color of the spots to change. I'll start with a list of colors to choose from:

```
@prop red.color_list {} rc

@edit red.color_list
enter
blue
green
yellow
purple
orange
brown
tan
black
white
.
save
done
```

Make up your own list of colors. It can be any length. Now, in the description verb, we're going to select one of these colors at random and put that color into the property `this.color`. Then we'll do the substitution:

```
@program red:description
base_description = pass(@args);
"Pick a random color for the spots.";
this.color = this.color_list[random($)];
return $string_utils:pronoun_sub(base_description);
.
```

The only new thing here is `this.color_list[random($)]`, and even that isn't completely new, because of our work with the `.moss_list` property on the pet rock. Working from the inside out, then. "$", when used inside the square brackets, refers to the number of elements in the list. That's why it doesn't matter how long your `.color_list` is. If you add more colors later, or remove some, changing the length of the list, the code will still work. `random($)` (again, when within square brackets) gives a random number between 1 and the length of the list. So, `this.color = this.color_list[random($)]` sets the property `this.color` equal to a random element of the list `this.color_list`.

Here's my whiz-bang fancy version – see if you can figure out what's going on here:

```
@rmprop red.color
@prop red.color1 "blue" r
@prop red.color2 "tan" r

@describe red as "You see a big red rock eater with
%[tcolor1] and %[tcolor2] spots."

@program red:description
base_description = pass(@args);
"Fancy version!  Two *different* colors of spots!";
index = random(length(this.color_list));
this.color1 = this.color_list[index];
this.color2 = (listdelete(this.color_list,
index))[random($)];
return $string_utils:pronoun_sub(base_description);
.
```

And so you see, unlike leopards, big red rock eaters *can* change their spots. And hopefully you have at least a glimmer of how this technique could be adapted to other situations. You could give your rock eater a variable number of heads, for example.

Is It a Boy or a Girl?

Being a critter, let's suppose that it has a gender. This step isn't strictly necessary, except that it allows me to use pronoun substitutions when typing in template messages for you to copy, and will give you some additional practice with pronouns as well. (Practice those pronoun substitutions!)

```
@chparent red to $gendered_object
```

What you get, by using this generic, is the verb for setting the gender, and a bunch of properties that are used for the various pronouns. Now you can set its gender, for example:

```
@gender red is male
```

Try typing:

```
@display red,
```

Note the comma, which means you want to display all inherited properties.

Pet the Nice Rock Eater...

We'll put a `:pet` verb on the rock eater, but this time we'll get a more gratifying response than we did from our rock. As always, we'll start simple and work up. Our initial goal will be to set up the `:pet` verb, and add message properties and their corresponding verbs:

```
@prop red.pet_msg "You pet %t."
@prop red.opet_msg "%N %<pets> %t."

@verb red:pet_msg tnt rxd
@addalias opet_msg to red:pet_msg

@program red:pet_msg
return $string_utils:pronoun_sub(this.(verb));
.

@verb red:pet this none none rxd

@program red:pet
player:tell(this:pet_msg());
player.location:announce(this:opet_msg());
.
```

This level of programming is so fundamental that I can almost type it in wholesale and have it work on the first try. (Though not quite -- I had a small typo in one of the verbs, and had to go back and fix it. Always test everything.) I usually set up the messages first, then the verb to do the pronoun substitution, then the verb that uses the messages.

So far, so good. But unlike a rock, a rock eater ought to react. So lets liven things up some. Here's a design decision: When it reacts, will the player and others in the room see the same thing, or different things? If I pet the rock eater, should I see, “The rock eater looks at you adoringly”, and others see, “The rock eater looks at

Yib adoringly”? Or is it okay if everyone (including me) sees, “The rock eater looks at Yib adoringly”? The second choice is easier. In the spirit of starting easy and getting fancy, we’ll do that. If the results aren’t satisfying, we can gussy things up some more, later.

But now I have a dilemma. My short-range plan is to add a message such as, “`%T thumps his tail happily.`” But my long-range plan is to have an optional list of possible responses, and maybe I could use that list for more than one thing (after he’s fed, for example). I want to name the message property in a way that is specific enough to be instructive to someone reading the code later, but general enough that I don’t have to limit myself to the `:pet` verb. I think I’m going to make it a happy response, and later, if the occasion arises, I can add unhappy responses and/or neutral responses.

```
@prop red.happy_response_msg "%T looks at %n adoringly." rc
@addalias "happy_response_msg" to red:pet_msg

@program red:pet
player:tell(this:pet_msg());
player.location:announce(this:opet_msg());
player.location:announce_all(this:happy_response_msg());
.
```

Test this. Notice the call to `player.location:announce_all`. There are three forms of the `:announce` verb on the generic room. `:announce` announces text to everyone except the player who typed the command. `:announce_all` announces text to everyone, *including* the player who typed the command. `:announce_all_but` takes an additional argument that specifies a list of objects *not* to see the text. We’ll use the third form later.

Now to diversify. Just as I’ve typed in a list of colors, I want to have a *list* of happy responses, and I want the message to select one of them. And, for compactness, I want to do it in the same message verb that I’m already using.

Here is the strategy: The `:pet_msg` verb is going to fetch `this.(verb)`, i.e. its corresponding message. If it’s a quoted string (that’s all we have, so far), then just do a pronoun substitution on that string. If it’s a list (of strings), then select one of them at random, and *then* do the pronoun substitution:

```
@program red:pet_msg
"If it's a list, pick one at random.";
"Then do the pronoun substitution.";
msg = this.(verb);
if (typeof(msg) == LIST)
  msg = msg[random($)];
endif
return $string_utils:pronoun_sub(msg);
.
```

First, we’ll test it to make sure all the old messages work fine. (Do that now.)

Then, edit red.happy_response_msg to be a list of strings:

```
@edit red.happy_response_msg
enter
%T thumps %[tpp] tail happily.
%T makes a rumbling noise in %[tpp] throat, reminiscent of a
cat's purring.
%T sighs contentedly.
%T does a happy little dance.
.
save
done
```

As with `%[tcolor]`, `$string_utils.pronoun_sub` translates `%[tpp]` into `this.pp`, which is a gendered object's *p*ossessive *p*ronoun.

Now, pet the nice rock eater to make sure that you get an appropriate variety of responses.

It Eats Rocks... Right?

We want to be able to feed rocks (and maybe other things) to the big red rock eater. Design decision: Shall it eat only rocks, or shall it eat anything, but especially like rocks? I choose to go for diversity on this one, so that you can feed the rock eater without having to hunt around for a rock. But either way, we'll need a strategy to tell whether an item *is* a rock or not, and there are some pitfalls there. Another design decision: Shall the rock eater eat players, or shall it be a domesticated rock eater that doesn't eat players? If it eats players, what happens to them then? I choose to sidestep that concern, and make a rock eater that eats rocks *and* other things, but doesn't eat players. (If I *were* going to have it eat players, I'd probably create a room that was the rock eater's tummy and move players there, where perhaps an adventure of some sort would await them. As always, I would start with something simple and make it progressively more complex.)

The syntax of the command will be, `feed <anything> to <rock eater>`. When the rock eater eats something, it will be moved to the rock eater itself. After a while, the food item will quietly go back to its home, or, if it doesn't have a home, to its owner's home. We have to account for the possibility that an item can't be moved to the rock eater (maybe its owner locked it down, for example), so we'll have messages to handle that case, and we have to account for someone *trying* to feed a player to the rock eater, and provide a suitable failure message. Accounting for every kind of misuse you can possibly think of is what makes for good, robust programming. We will have many opportunities to practice our pronoun substitution.

First, I'm going to tackle some behind-the-scenes stuff, in particular, filtering what things the rock eater can eat.

Baseline check – Do you still have your pet rock handy?

```
@move rock to red
```

You should get a message to the effect that either your rock doesn't want to go, or the big red rock eater didn't accept it.

```
@verb red:acceptable tnt rxd

@program red:acceptable
"This verb returns a truth value if an item may be moved to
the rock eater, and 0 if an item may not be moved to the
rock eater.";
{item} = args;
if (is_player(item))
  "No players!";
  result = 0;
else
  result = 1;
endif
return result;
.
```

`{item} = args;` This is a special kind of assignment statement. This verb won't be called from the command line. But it needs to receive some information (called *arguments*) so that it knows *what* is under consideration for acceptance. The built-in variable `args` is a list of arguments. We only expect one argument to this particular verb. The form `{item} = args`, is the preferred way of writing, `item = args[1]`. This is an idiom of the language; it's detailed in section 4.1.9 of the programmer's manual.

Here's one of my philosophies of writing code: Nobody writes bug-free code. All code will be maintained sooner or later. Even the person who writes the code sometimes forgets what e was thinking when e wrote it. It is better to write longer code that is easy to understand than to write compact code that is cryptic. If and only if you can compact the code without undue sacrifice of clarity, then more compact code is better.

Now, a shorter, more compact version:

```
@program red:acceptable
"This verb returns a truth value if an item may be moved to
the rock eater, and 0 if an item may not be moved to the
rock eater.";
"Players aren't accepted.";
{item} = args;
result = (is_player(item) ? 0 | 1);
return result;
.
```

And a shorter version still:

```
@program red:acceptable
"This verb returns a truth value if an item may be moved to
the rock eater, and 0 if an item may not be moved to the
```

```
rock eater.";
"Players aren't accepted.";
{item} = args;
return (is_player(item) ? 0 | 1);
.
```

Now try teleporting your rock to the rock eater. This should work. If you look at Red, you shouldn't see the rock, which is fine, since tummies are (usually) opaque. But is the rock really in there? Type:

```
@contents red
```

to see. This will also remind you of the object number of your rock, in case you want to teleport it back out again.

If you wanted to make a rock-shaped bulge in its tummy (say), you might alter its :description verb and/or add a verb called `:tell_contents`, which is what containers and rooms do. You would check the value of its `.contents` property and go from there. (The details are left as an exercise for the intrepid new programmer.)

Now, to the business of seeing to it that stomach contents get returned eventually.

The verb `:acceptable` is expected to return a truthful, silent answer, yes or no, to the question, "Will object A accept object B?" The verb `:accept` does the actual business of accepting (or rejecting), and may do any associated processing. For example, if you have ever tried to join someone who was in a locked room, you got a message saying that either you didn't want to go, or the room didn't accept you. That's done by the `:accept` verb. The `:accept` verb on the big red rock eater is going to fork a task to move the incoming item back to some appropriate place at a later time (if the item is acceptable). If the item is not acceptable (if it's a player, for example), then the `:accept` verb will just return a value of 0.

I had to tinker with the :accept verb quite a lot before it worked to my satisfaction, so I'll spare you the play-by-play development process. Don't worry if you don't understand every detail, but do try to follow along. Later you might want to write a custom `:accept` verb on some other object, and you'll know to revisit this example for a deeper understanding of it.

```
@prop red.digestion_duration 30 rc
```

This is the duration (in seconds) that a thing will stay in Red's tummy. While I'm testing, I'll set it to something short, like 30 seconds. When I'm satisfied that everything works, I'll change it to something like an hour, maybe.

```
@prop red.return_item_home_msg "The housekeeper arrives and
drops off %[titem]." rc
@prop red.item "This will be set to item.name by the :accept
verb." r
@addalias "return_item_home_msg" to red:pet_msg

@verb red:accept tnt rxd
```

```
@program red:accept
"Hold an item (while digesting), then try to send it home.";
{item} = args;
if (result = this:acceptable(@args))
  fork (this.digestion_duration)
    "Is it still there?";
    if (item.location == this)
      "Figure out where to send it, and try to send it
there.";
      place = ($object_utils:has_property(item, "home") ?
                item.home |
                item.owner.home);
      item:moveto(place);
      "Now see if it actually arrived.";
      if (item.location == place)
        if ($object_utils:has_verb(place, "announce_all"))
          "Set up item.name for appropriate pronoun
substitution.";
          this.item = item.name;
          place:announce_all(this:return_item_home_msg());
        endif
      else
        "We failed to get rid of it gracefully, just get rid
of it.";
        this:eject_basic(item);
      endif
    endif
  endfork
endif
return result;
.
```

There is one subtlety in particular to which I would like to call your attention. In the statement:

```
if (result = this:acceptable(@args))
```

I have done an assignment statement *within* the parenthetical conditional statement. This is perfectly legal and is often done. It's the same as:

```
if (this:acceptable(@args))
  <do stuff>
endif
return this:acceptable(@args);
```

except that I call the `:acceptable` verb once instead of twice, saving the result for later.

And now (at long last) we are ready to write the `:feed` verb itself.

Chow Time!

After all we've been through, this part will be quite easy. Here's the prototype:

```
@verb red:feed any to this rxd

@program red:feed
dobj:moveto(this);
if (dobj.location == this)
  player:tell("Red chomps hungrily.");
else
  player:tell("Red looks at you dubiously.");
endif
.
```

The item being fed to Red is the direct object of the command, and its object number is stored in the built-in variable `dobj`. Red is the indirect object of the command, and its object number will be stored in the built-in variable `iobj`, as well as the built-in variable `this` (the object on which the currently-executing verb is defined).

Here is the cleaned up, robust version:

```
@prop red.feed_msg "You feed %d to %t." rc
@prop red.ofeed_msg "%N %<feeds> %d to %t." rc
@prop red.no_feed_msg "You try to feed %d to %t." rc
@prop red.ono_feed_msg "%N %<tries> to feed %d to %t." rc
@prop red.ptui_msg "%T looks at %n dubiously. . o O ( Ptui!
)" rc

@addalias feed_msg to red:pet_msg
@addalias ofeed_msg to red:pet_msg
@addalias no_feed_msg to red:pet_msg
@addalias ono_feed_msg to red:pet_msg
@addalias ptui_msg to red:pet_msg

@program red:feed
"Try to feed it something.";
dobj:moveto(this);
if (dobj.location == this)
  "It ate the whole thing.";
  player:tell(this:feed_msg());
  player.location:announce(this:ofeed_msg());
  this.location:announce_all(
    this:happy_response_msg());
else
  "Ack!  Ptui!  Wouldn't accept it.";
  player:tell(this:no_feed_msg());
  player.location:announce(this:ono_feed_msg());
  this.location:announce_all(this:ptui_msg());
endif
.
```

Have you been testing all along? This works pretty darned well. Except that I tried to feed my pet rock to the rock eater before it had been returned again (I test a lot), and I got a traceback:

```
#26703:feed, line 2:  Invalid indirection
(End of traceback)
```

I can reproduce the error by trying to feed Red an object that doesn't exist:

```
feed mother-in-law to red
```

```
#26703:feed, line 2:  Invalid indirection
(End of traceback)
```

Looking at line 2, we're trying to move `dobj`. Aha! We need to add a check to make sure that the specified direct object is a *valid* object:

```
@program red:feed
"Try to feed it something.";
if (valid(dobj) && (dobj.location in {player,
player.location}) && (this.location in {player,
player.location}))
  dobj:moveto(this);
else
  "Either the thing being fed or the rock eater is not in
the vicinity, or the thing being fed isn't a valid object.";
  player:tell("I don't see that here.");
  "Just quit this verb right away.";
  return;
endif
if (dobj.location == this)
  "It ate the whole thing.";
  player:tell(this:feed_msg());
  player.location:announce(this:ofeed_msg());
  this.location:announce_all(this:happy_response_msg());
else
  "Ack!  Ptui!  Wouldn't accept it.";
  player:tell(this:no_feed_msg());
  player.location:announce(this:ono_feed_msg());
  this.location:announce_all(this:ptui_msg());
endif
.
```

I chose not to extract, "I don't see that here," into its own message. It isn't something I ever expect to change; I don't need to do any pronoun substitution; and it does double duty as a comment within the code. Also during play testing, I found out that someone could feed Red remotely, and I don't want that because the messages display to the wrong place and seem incongruous, so I added a check to make sure that the thing being fed and the rock eater were either in the player's possession, or in the same location as the player. I should go back and add that same check to the `:pet` verb, too. I might not have found this bug on my own. Someone else noticed it. I like to invite friends to play-test my objects before I present them to the public, because that helps me find and fix the bugs I *didn't* think to look for.

```
@program red:pet
if (this.location in {player, player.location})
  player:tell(this:pet_msg());
  player.location:announce(this:opet_msg());
else
  player:tell("I don't see that here.");
endif
.
```

As a last little fillip, I offer the following: Since our `pet_msg` verb can handle a string *or* a list, I'm going to edit `red.ptui_msg` for greater variety:

```
@edit red.ptui_msg
enter
%T takes one taste of %d and promptly spits %[dpo] out
again. . o O ( Ptui! )
%T eats %d, but then, with a look of great consternation on
%[tpp] face, acks %[dpo] back up again. . o O ( Ptui! )
 .
save
done
```

(`%[dpo]` is the direct object's object pronoun.)

Whew, I'm hungry! I think I'll have a snack before going on to the next part.

The Breath of Life

The last step in making a pet is to enable it to respond spontaneously to things that happen around it, and thus seemingly take on a life of its own.

Whenever a verb calls a room's `:announce` verb (or one of its variants), the `:announce` verb calls the `:tell` verb on every object in the room that has one, and sends the specified text in as an argument. So by adding a `:tell` verb to our rock eater, it will automatically start "hearing" things going on around it. And then it can respond. We'll put in a random delay to make its actions seem even more independent:

```
@prop red.response_delay 20 rc
```

A delay (in seconds).

```
@prop red.action_msg {} rc
@edit red.action_msg
```

```
enter
%T chases %[tpp] tail in a slow, circling ballet.
%T leaps into the air and does a back flip, in a comic bid
for attention.
%T snuffles around, looking for rocks.
%T looks at you with big, sad, soulful eyes.
%T makes a wurfling sort of noise.
.
save
done

@addalias action_msg to red:pet_msg

@verb red:tell tnt rxd

@program red:tell
fork (random(this.response_delay))
    this.location:announce_all(this:action_msg());
endfork
.
```

Now say, “Boo,” or something. You can check on your forked tasks by typing `@forked`.

Whoa! Down Boy!

```
@kill red:tell
```

Well! By now you’ve discovered that once started, Red just won’t quit. This is because Red hears Red’s own text, and responds to it! So what you get is a sort of chain reaction. Really, this is too much of a good thing, so now we have to work on toning things down some.

```
@program red:tell
fork (random(this.response_delay))
  this.location:announce_all_but({this},this:action_msg());
endfork
.
```

First, we’ll use that third form of `:announce` that I mentioned earlier, `:announce_all_but`. This way you won’t get an endless chain of actions. But you’ll still get an action out of Red every single time there’s a noise in the room. So for my next trick, I’m going to make it so that sometimes he responds, and sometimes he doesn’t:

```
@prop red.action_odds 3 rc

@program red:tell
```

```
if (random(this.action_odds) == 1)
  fork (random(this.response_delay))
    "Odds of responding are 1 in this.action_odds.";
    this.location:announce_all_but({this},
this:action_msg());
  endfork
endif
.
```

This is better. But you might want a quieter pet still. (I did.) Whenever I would pet or feed Red, the messages would trigger an additional response, which still seemed like too much. So I can go back and edit the `:pet` and `:feed` verbs, *or* I can tinker with the `:tell` verb a bit more and prevent Red from hearing any of his own messages in the first place. Take a quick look at `help callers()`. This built-in function returns a list of all the object/verb pairs (tuples, actually -- there's additional information) that resulted in the current verb being called. So we're going to take a look at `callers()` from within `red:tell` and filter out any noise generated by Red himself. This is pretty advanced stuff, and even I do a bit of preliminary prototyping before using `callers()`, because I always forget just how it goes. Here's the prototype:

```
@prop red.callers 0 r
```

A temporary property to hold data that I want to look at.

```
@program red:tell
this.callers = callers();
if (random(this.action_odds) == 1)
  fork (random(this.response_delay))
    this.location:announce_all_but({this},
this:action_msg());
  endfork
endif
.
```

`Pet red`, to trigger the process. Now we will use a special form of the `eval` command, "#" to inspect the results:

```
#red
=> #26703 (Big Red Rock Eater)

#red.callers
=> {{#23920, "announce_all", #2, #3, #58337}, {#23920,
"announce_all", #24442, #17755, #58337}, {#23920,
"announce_all", #61050, #9805, #58337}, {#26703, "pet",
#58337, #26703, #58337}}
```

By scrutinizing it, I find the number of *my* rock eater (#26703 in this example). Your numbers will be different, but look anyway. This is list of lists. I want to consider only the first elements, and see if Red's `:tell` verb was indirectly called by

Red. If it wasn't, then and only then will Red react. To do that, I'll use `$list_utils:slice` (there's help text – give it a try):

```
@program red:tell
"Respond to noises generated by anything  *except* this.";
if (random(this.action_odds) == 1)
  if (!(this in $list_utils:slice(callers())))
    fork (random(this.response_delay))
        this.location:announce_all_but({this},
          this:action_msg());
    endfork
  endif
else
  "This was the source of the noise, so do nothing.";
endif
.

@rmprop red.callers
```

Test this by saying things, petting your rock eater, feeding it, etc. Check `@forked` a lot.

This is almost perfect. (Wouldn't you know.) *If* you were to trigger Red's `:tell` verb and fork the task, then move Red to `#-1`[19], you would get a traceback, because `#-1` doesn't have an `:announce_all_but` verb. So we'll add a check for that:

```
@program red:tell
"Respond to noises generated by anything  *except* this.";
if (!(this in $list_utils:slice(callers())))
  if (random(this.action_odds) == 1)
    fork (random(this.response_delay))
      if ($object_utils:has_verb(this.location,
"announce_all_but"))
        this.location:announce_all_but({this},
this:action_msg());
      endif
    endfork
  endif
endif
.
```

Last, I take a step back to see if I can condense the code without sacrificing clarity, and I think I can. Those two nested `if` statements can be consolidated into one:

```
@program red:tell
"Respond to noises generated by anything  *except* this.";
if (!(this in $list_utils:slice(callers())) &&
```

[19] `#-1` is the null object. It has no properties or verbs.

```
    (random(this.action_odds) == 1))
  fork (random(this.response_delay))
    if ($object_utils:has_verb(this.location,
"announce_all_but"))
      this.location:announce_all_but({this},
this:action_msg());
    endif
  endfork
endif
.
```

Now all that's left is to tinker with `red.action_odds` and `red.response_delay` until you get the right feel for *your* pet. Some rock eaters might be placid, others might be frisky.

Care and Feeding of a Big Red Rock Eater

Just a last little bit of grooming to do, and we're done:

```
@prop red.obvious_verbs {} rc

@set red.obvious_verbs to {}
@edit red.obvious_verbs
enter
  pet %<what>
  feed <anything> to %<what>
.
save
done

@verb red:examine_verbs tnt rxd

@program red:examine_verbs
what = dobjstr;
vrbs = {};
for vrb in (this.obvious_verbs)
  vrbs = {@vrbs, $string_utils:substitute(vrb, {{"%<what>",
what}})};
endfor
return {"Obvious verbs:", @vrbs};
.
```

This is the usual stuff. I deliberately omitted `@gender %<what>` from the list of examine verbs, because that's an owner-only verb: You know it's there, and no one else needs to.

```
@prop red.help_msg {} rc

@edit red.help_msg
enter
This is %t.  Treat %[tpo] with love and kindness and %[tps]
will be your friend forever.

%[Tps] likes to eat rocks, but will eat just about anything
except players.  (Things which have been eaten will be moved
back to their homes after a while.)
.
save
done

@verb red:help_msg tnt rxd

@program red:help_msg
"This message has its own separate verb because the regular
message verb returns a random element of a list.  But if
this.help_msg is a list, we want the whole thing.";
return $string_utils:pronoun_sub(this.(verb));
.
```

This is your big red rock eater. Treat it with love and kindness, and it will be your friend forever.

Afterword

That’s about it.

We started with the very rudiments of adding a verb to an object, and have worked our way through some fairly sophisticated stuff. You may or may not feel that you’ve grasped it all, but my primary goal was to give you some exposure to how MOOcode works, and to make it all less intimidating, in hopes that you will be inspired to explore further.

Be bold! Experiment. Try things. Don’t be afraid of breaking something. There is very little on the MOO than can be harmed by accident, and even less that can’t be fixed. Go for it!

MOO Programming Reference

This section is geared toward people who are at least somewhat comfortable with programming. For the fine points I refer you to the programmer's manual itself. Here, I have tried to present an overview of the MOO programming language in a format that is biased towards ease of reference rather than exhaustive explication. I have provided brief explanations of some points where I think clarification might be helpful.

Data Types

Variables are not of fixed data type; they become the data type of the value assigned to them. Use `typeof(<variable>)` to see what you've got.

`INT`	Integer: `29, 0, -0, 5`
`FLOAT`	Floating point number: `29.0, 0.0, 5.0`
`NUM`	Historical, same as `INT`. (`FLOAT` was a later addition.) `FLOAT`s and `INT`s don't mix and match. Use toint() or tofloat() to force one to be the other. Note that (1 == 1.0) evaluates to false.
`STR`	A quoted string: `"pickles"`. To include the double quote mark itself in a string, precede it with the backslash character: `"Oliver shouts, \"Yow!\""` Strings are case-insensitive: `("carrot" == "CarROT")` evaluates to true. Use `equal("carrot", "CarROT")` if you need to differentiate between the two.
`OBJ`	An object, e.g: `#1234`. In conditional statements, an object by itself evaluates to false. Use `valid(<object>)` instead. Some special objects: • `#-1` or `$nothing`. Not valid, but anything may be moved there. The canonical invalid object. • `#-2` or `$ambiguous_match`. • `#-3` or `$failed_match` • `$garbage` This object and kids of it are valid but not useful. They are all owned by the special system player Hacker. When an object is recycled, it becomes a kid of `$garbage`.

LIST	`{1, 2.5, "a string", {"a", "sublist"}, #321, E_RANGE, 2.5}`
	
	`LIST`s are designated with curly braces (`{}`), may nest to an arbitrary number of levels, and their elements need not be of the same data type. Their elements are preserved in order and may include duplicates (i.e. they are not like mathematical sets).
	
	The @ operator yields the elements of the list as separate elements. Another way to phrase this is that it is the inverse of putting curly braces around some elements. Two canonical uses:
	
	`<some-list> = {@<some-list>, <new_element>};` This is the usual way to add an element onto the end of a list. If `a` is `{1, 2, 3}`, and `b` is 4, then `{@a, b}` gives the four-element list `{1, 2, 3, 4}`, whereas `{a, b}` would give the two-element list `{{1, 2, 3}, 4}`.
	
	`pass(@args);` This causes the identically-named verb on the current object's parent to be run with the same argument list. If you just did `pass(args)`, then the arguments to the verb being called would have an extra set of braces around them, thus making the argument list `{{a, b, c}}`, for example, instead of `{a, b, c}`.
	
	`<element> in <some-list>` will return the (1-based) index of the first instance of `<element>` in `<some-list>` if it is present, `0` otherwise. A typical usage might be:
	
	`if (item.location in {player,`
	` player.location})`
	`  <exprs>;`
	`endif`

ERR	E_NONE	no error
	E_TYPE	type mismatch
	E_DIV	division by zero
	E_PERM	permission denied
	E_PROPNF	property not found
	E_VERBNF	verb not found
	E_VARNF	variable not found
	E_INVIND	invalid indirection
	E_RECMOVE	recursive move
	E_MAXREC	too many verb calls (max recursion)
	E_RANGE	range error (subscript too large, or zero, or negative)
	E_ARGS	incorrect number of arguments
	E_NACC	move refused by destination (i.e. object not acceptable)
	E_INVARG	invalid argument
	E_QUOTA	resource limit exceeded
	E_FLOAT	floating-point arithmetic error

Errors can be raised (yielding a traceback) or caught (then handled or ignored). The following two constructs are used to trap errors and deal with them:

```
`<expr1> ! ANY => <expr2>'
```

See section 4.1.12 of the programmer’s manual. The single quotes are part of the expression, and are specifically back single quote and forward single quote.

```
try
  <exprs>;
except ANY
  <alternate exprs>;
endtry
```

See sections 4.2.7 and 4.2.8 of the programmer’s manual.

Subscripting

Everything is 1-based.

You can subscript lists or strings.

`<list-or-string>[<expr1>..<expr2>]` gives slices (sub-list or sub-string). `<expr1>` and `<expr2>` must be in range; `<list-or-string>[$]` is the last element or character.

The following are well-formed:

```
<variable> = <list-or-string>[<expr>];
<variable> = <list-or-string>[<expr1..expr2>];
<list-or-string>[<expr1>] = <expr2>;
<some-list>[<expr1>..<expr2>] = <expr3>;
<some-string>[<expr1>..<expr2>] = <expr4>;
```

Note that in the above example, `<expr3>` must evaluate to a list and `<expr4>` must evaluate to a string.

Accessing Properties and Verbs on Objects

`$<something>` is the same as `#0.<something>`.

You can use parentheses to access property names and verbs dynamically:

```
<object>.(<calculated-property-name>)
<object>:(<calculated-verb-name>)
```

Variables

Variables are local and dynamic, coming into existence when assigned a value. For global variables, define and use a property.

In addition to the data types themselves (which evaluate to integers), the following built-in variables are provided:

`this`	The object on which the currently-running verb is defined.
`player`	The object number of the player who typed in the command.
`caller`	The objnum of the object on which the calling verb is defined, or `player`, if the verb was called from the command line.
`verb`	The name by which the currently-running verb was invoked. Verbs may have aliases.
`args`	The list of arguments with which a subroutine was called or, if a command-line verb, with which the command was invoked.
`argstr`	Everything that was typed in after the verb name on the command line.
`dobj`	The direct object as parsed from the command line.
`dobjstr`	The string from which dobj was matched.

`prepstr`	The string that was parsed as the preposition.
`iobj`	The indirect object as parsed from the command line.
`iobjstr`	The string from which iobj was matched.

Any of these may be reassigned within a verb and their values will persist into the next verb call except that `caller` will change to the current object, and a changed value of `player` will not persist unless the verb's owner is a wizard.

Scattering Assignment

Several variables may be assigned values in a single line, and this is often done to assign incoming arguments to named variables. A typical example might be:

```
{who, what, ?where = player.location, ?when=time()} = args;
```

See section 4.1.9 of the programmer's manual for a detailed explanation.

Operators

The following operators apply, in order of precedence:

`!` `-`	not arithmetic negation (without a left operand)
`^`	exponentiation

`*`
`/`
`%`
	multiplication
	division
	modulo

`+` `-`	addition (note, + also concatenates two strings) subtraction

`==`
`!=`
`<`
`<=`
`>`
`>=`
`in`
	is equal to (note, easy to confuse with the assignment operator – nasty!)
	is not equal to
	less than
	less than or equal to (note, =< doesn't work)
	greater than
	greater than or equal to (note, => doesn't work)
	element position in a list

`&&`	logical "and"

`||`	logical “or”

`… ? … | …`	the conditional operator.
	`<expr1> ? <expr2> | <expr3>`
	is equivalent to:
	`if (<expr1)`
	`  <expr2>;`
	`else`
	`  <expr3>;`
	`endif`

`=`	assignment (note, easy to confuse with a test for equality – nasty!)

Assignments may appear within expressions. Use parentheses liberally to avoid mistakes and confusion.

Truth Values

`0`, `-0`, `0.0`, `-0.0`, `""`, `{}`, errors, and objects all evaluate to false. Anything else evaluates to true.

Compound Statements

Use a semicolon after expressions within the body of a compound statement, but not after lines of the compound statement itself, thus:

```
if (<expr1>)
  "This is a comment.";
  <expr2>;
  <expr3>;
elseif (<expr4>)
  <more-exprs>;
elseif (<expr5>)
  <something else entirely>;
else
  "None of the above.";
  <final-exprs>;
endif
```

Looping

```
for <variable> in (<some-list>)
  <exprs>;
endfor

for <index> in [<int1>..<int2>]
  <exprs>;
endfor

while (<condition>)
  <exprs>;
endwhile

break;

break <name>;

continue;

continue <name>;
```

(See section 4.2.5 of the programmer's manual for the fine points of `break` and `continue`.)

Background Tasks

```
fork (<delay-in-seconds>)
  <exprs>;
endfork

fork <variable> (<delay-in-seconds>)
  <exprs>;
endfork
```

In the second example, `<variable>` receives the number (task_id) of the forked task.

Time Management

A task is the execution of a command from start to finish, or the execution of the statements within a `fork/endfork` statement from start to finish. Tasks are identified with numbers, and are allotted a fixed number of ticks and a fixed number of seconds for execution. The LambdaCore default is 30,000 ticks for a foreground task and 15,000 ticks for a background task. Properties to override these numbers may be added to the object `$server_options` by a wizard and inspected (if present) by programmers.

If a task runs out of ticks, it is unceremoniously terminated by the system. If a task is in danger of running out of ticks, a programmer may get a new allotment by suspending the task briefly (note, `suspend ($login.current_lag)` is considered polite). When a task suspends, it obtains an additional allotment of ticks, so it is not uncommon to find or place a `suspend()` statement either right before or inside of a loop.

The utility verb `$command_utils:suspend_if_needed()` might suggest itself, but in fact it uses up a fair number of ticks, itself. Current fashion is to use a line of the form:

```
((ticks_left() < 3000) && suspend($login.current_lag));
```

See also sections 4.4 and 5.2.8 of the programmer's manual.

Argument Specifiers

When defining a verb on an object (with the `@verb` command), you must provide specifiers for the arguments with which the verb will be invoked.

The allowable specifiers are:

- direct object specifiers

```
this
any
none
```

- prepositions

```
none
any
with/using
at/to
in front of
in/inside/into
on/onto/upon/on top of
from/from inside/out of
over
through
under/underneath/beneath
behind
beside
for/about
is
```

```
as
of/off of
```

- indirect object specifiers

```
this
any
none
```

Definite and indefinite articles are omitted. When deciding which argument specifiers to use, it is helpful to imagine what a user would actually type when invoking the command, then generalize from that. When writing subroutines that aren't intended to be invoked from the command line, specify the arguments as "`this none this`".

Eval

The `eval` command evaluates a string as MOOcode. Like `say` and `emote`, it can be abbreviated to a single character command, `;`. A second form, `;;` evaluates a sequence of expressions, each terminated by a semicolon (as in a verb). Compound statements don't end in a semicolon. The form using two semicolons prints out 0 as its value; if you want to see results you should include a call to `player:tell();` at the end:

```
;;"Count players who have more than ten aliases"; total = 0;
for dude in (players()) if (length(dude.aliases) > 10) total
= total + 1; endif endfor player:tell("Total:  " +
tostr(total));
```

A second form of `eval`, "#" matches an object by name if it's in your vicinity, and is useful for looking at properties or just quickly finding out the object number of something close by. Property names can be chained:

```
#rock
#rock.moss_list
#yib p
#yib.aliases p
#yib.location.owner.name p
```

The last form, terminated by " `p`" matches the name of a player even if you're not in eir vicinity, so that you don't have to know eir number to look at a (readable) property on em. "#" can also be used with object numbers directly:

```
#58337.location.contents
```

Use `@setenv` to set up some commonly-used variable settings in advance:

```
@setenv me = player; here = player.location;
```

Inspect the result with

```
#me.eval_env
```

See also `help eval` and `help #`.

Ownership and Permissions

Every object has an owner. Every property on every object has an owner, but it doesn't have to be the same as the owner of the object. Every verb on every object has an owner, but it doesn't have to be the same as the owner of the object.

Task permissions are expressed as an object number, that of the player who owns the verb currently being executed.

The function `caller_perms()` returns the task permissions of the calling verb, or `#-1` (an invalid object) if the currently running verb was called from the command line.

Inherited verbs always have the same owner as the owner of the corresponding verb on the parent or ancestor object. They run with that owner's permissions, except that wizard-owned verbs can set the task permissions to another (usually non-wizardly) player.

To `+c` Or To `–c`, That Is The Question

Ownership of an inherited property *depends* on whether the property was initially defined as `+c` or `-c`. If it was defined as `+c` (think "may be *changed* by the owner of the child/descendent object"), then the property is *owned by* the owner of the child/descendent object. If the property was initially defined as `-c` then the property on all children and descendents is owned by the player who defined the property on the parent/ancestor object, *and its value can be changed by verbs running with that player's permissions*. This becomes relevant when one is making a generic object. If the owner of a child or descendent object will need to `@set` or otherwise change the property, then define it as `+c`. This is typically done for messages, and also for other parameters, for example the number of times one must turn the crank before the jack in the box pops out. If, on the other hand, one of the verbs you write on the generic will need to change the value of a property, then it should be defined as -c so that the property on all descendent objects will still be owned by you. Then your verbs, running with your permissions, can change it (for example, the number of times the crank on the jack in the box has been turned so far).

When you make an object strictly for your own use, it really doesn't matter whether the properties are `+c` or `-c`. It becomes an issue when other people make kids of your object. Then if a property that one of your verbs needs and tries to change is mistakenly `+c`, the verb will encounter a permissions error. If you `@chmod` the property to `-c`, then all *new* kids of the object will have that property owned by you, but it isn't changed retroactively for existing kids. If you make a property +c and find out later that it should have been -c, you can change it on all descendents by evaluating the following:

```
;$wiz_utils:set_property_flags(<object>, <property_name>,
<property_flags>)
```

You don't have to be a wizard to use this verb – just the owner of the object. Here's an illustration, supposing that a generic conker is object #1234:

```
;$wiz_utils:set_property_flags(#1234, "thwaps", "r")
```

This would have the effect of making the `.thwaps` property on all descendents of the generic conker readable but neither writable nor changeable (by owners of kid objects), and the property would be owned by the author of the generic conker in all cases (and could be changed by that player's verb(s)).

Hidden Treasures

Some verbs are called automatically, seemingly invisibly. Here are some of them:

`<object>:look_self`	Called when you look at `<object>`
`<object>:description`	Called (if it exists) by `:look_self`
`<object>:tell_contents`	Called (if it exists) by `:look_self`
`<object>:enterfunc`	Called when something is moved to `<object>`
`<object>:exitfunc`	Called when something is removed from `<object>`'s `.contents`.
`<room>:confunc`	Called when someone connects inside a room.
`<room>:disfunc`	Called when someone disconnects inside a room.
`<player>:confunc`	Called when a player connects
`<player>:disfunc`	Called when a player disconnects
`<object>:initialize`	Called when an object is created. Use, for example to initialize parameters on the kid of a generic object.
`<object>:recycle`	Called right before an object is recycled.
`<player-or-room>:huh`	Called if the parser can't find an object with the appropriate verbspec. This is how exits in rooms are invoked without the exit objects' having to be *in* rooms, for example.

A Couple “Tricks of the Trade”

Sending mail messages from within a verb: The relevant verb is `$mail_agent:send_message`. Personally, I always find the help text hard to read, so I am providing this illustrative example, which I hope may be helpful:

```
;$mail_agent:send_message(me, {me},
  "This is the subject heading", {"Line1", "Line2", "",
  "Oooga boooga!"})
```

Creating objects on the fly is fun, and possible if you are not over quota. Here is an example of how it’s done.

```
@verb me:test none none none rd

@program me:test
"Sample verb to demonstrate creating an object on the fly.";
thing1 = `$recycler:_create($thing) ! ANY =>
  $nothing';
if (valid(thing1))
  thing1:set_name("thing1");
  thing1:moveto(player.location);
  "If you create a lot of things, then you need to measure
them as you go to avoid a 'resource limit exceeded' error.";
  $quota_utils:object_bytes(thing1);
  player:tell("You now have something that you didn't have
before!");
else
  player:tell(
    "Couldn't create thing1.  Don't know why.");
endif
.

test
```

To recycle an object (that you own) from within a verb:

```
$recycler:_recycle(<object>);
```

Programming Feature Objects

This is a very brief summary of the steps involved in creating a feature object. It isn’t a tutorial on programming in general, but highlights a couple of quirks associated with programming this particular kind of object.

First, create a kid of the generic feature object:

```
@create $feature named <your-FO-name>
```

Then describe it.

Then program some verbs on it. Note that the verbs have to have the "`x`" permission flag set, so that they can be called from other verbs.

Then add help text. This can be done in either of two different ways. The first is to edit your feature object's `.help_msg` property. You should present each of the verbs on your FO that are intended for public use (as opposed to internal subroutines), give the syntax for the verb's usage, and a brief explanation of what the verb does. The other way is to put the documentation for each verb intended for public use as a set of comments at the top of the verb. The second is the officially preferred method (as per `help $feature`), but both will work.

THEN: In either case you must edit your FO's `.feature_verbs` property. If you put all the documentation in the help_msg property, then type:

```
;<your-FO>:set_feature_verbs({})
```

If you put the documentation for each public-use verb at the top of each verb, then type:

```
;<your-FO>:set_feature_verbs({"<verb1>", "<verb2>", ... ,
"<last-verb>"})
```

If you wish to restrict who may add your feature object, write a custom `:feature_ok` verb on it. This verb should return 0 if for whatever reason the person may not add the feature, or a truth value otherwise. An example of when this might come in handy might be a feature only for use by wizards.

See also `help features` and `help $feature`.

Built-In Functions

See the online help text or the programmer's manual for the specifics of each individual function – here's what's there:

The quintessential object-oriented function:

```
pass()
```

General operations applicable to all values:

```
typeof()                           toobj()
tostr()                            tofloat()
toliteral()                        equal()
toint()                            value_bytes()
tonum()                            value_hash()
```

Operations on Numbers:

```
random()                           max()
min()                              abs()
```

```
floatstr()                          cosh()
sqrt()                              tanh()
sin()                               exp()
cos()                               log()
tan()                               log10()
asin()                              ceil()
acos()                              floor()
atan()                              trunc()
sinh()
```

Operations on Strings:

```
length()                            match()
strsub()                            rmatch()
index()                             substitute()
rindex()                            crypt()
strcmp()                            string_hash()
decode_binary()                     binary_hash()
encode_binary()
```

Operations on Lists:

```
length()                            listdelete()
is_member()                         listset()
listinsert()                        setadd()
listappend()                        setremove()
```

Manipulating Objects:

```
chparent()                          verb_args()
valid()                             set_verb_args()
parent()                            add_verb()
children()                          delete_verb()
object_bytes()                      verb_code()
max_object()                        set_verb_code()
move()                              disassemble()
properties()                        players()
property_info()                     is_player()
set_property_info()                 set_player_flag()
add_property()                      connected_players()
delete_property()                   connected_seconds()
is_clear_property()                 idle_seconds()
clear_property()                    notify()
verbs()                             buffered_output_length()
verb_info()                         read()
set_verb_info()                     force_input()
```

`flush_input()`	`connection_option()`
`output_delimiters()`	`open_network_connection()`
`boot_player()`	`listen()`
`connection_name()`	`unlisten()`
`set_connection_option()`	`listeners()`
`connection_options()`	

Operations Involving Times and Dates:

```
time()
ctime()
```

MOO-Code Evaluation and Task Manipulation:

`raise()`	`task_id()`
`call_function()`	`suspend()`
`function_info()`	`resume()`
`eval()`	`queue_info()`
`set_task_perms()`	`queued_tasks()`
`caller_perms()`	`kill_task()`
`ticks_left()`	`callers()`
`seconds_left()`	`task_stack`

Administrative Operations:

`server_log()`	`dump_database()`
`renumber()`	`db_disk_size()`
`reset_max_object()`	`shutdown()`
`memory_usage()`	

$Utils

Some of the built-in functions are used frequently in everyday programming, some are used rarely, or only by wizards, or both. The MOO also provides a collection of utilities packages. Each utilities package has its own top-level help text, and each verb has more detailed help text. This list is just the $utils packages available in LambdaCore. A reference list of all the verbs on each is provided in Appendix B.

`$wiz_utils`	`$lock_utils`
`$math_utils, $trig_utils`	`$list_utils`
`$set_utils`	`$command_utils`
`$seq_utils`	`$code_utils`
`$gender_utils`	`$building_utils`
`$time_utils`	`$string_utils`
`$match_utils`	`$generic_utils`
`$object_utils`	`$quota_utils`

```
$byte_quota_utils                  $matrix_utils
$object_quota_utils                $convert_utils
```