Chapter 3 – What's Going On, Here?

Objects

MOO stands for "MUD, Object-Oriented"[13]. This section discusses in depth what an object actually is.

Objects are the building blocks of a MOO. They are things; in a sense, they are nouns: A noun is a word that represents a person, place, or thing; an object is a data construct within the computer program that is the MOO (i.e. the server) that represents a person, place, or thing. Some nouns represent concrete things, such as chairs, cats, and candy, while others represent intangible things, such as news, knowledge, and abilities. Likewise, some objects represent concrete things within the MOO (chairs, cats, candy) while other objects represent intangible things (news, knowledge, and abilities). But intangible things are still things, and therein lies the nature of an object.

Every object in a MOO is assigned a number upon creation. This number is unique within the MOO and immutable. If absolutely every characteristic of an object were changed – its name, its owner, its location, its description – its number would still be the same.

There are certain pieces of information that are attached to every valid object without exception. (An object that has a number but doesn't have these pieces of information associated with it is an invalid object, by definition.) These pieces of information include the object's owner (itself identified by object number), its location (identified by object number), its contents (a list of one or more object numbers), and its parent (identified by object number).

Object parenthood is a special concept. An (imperfect) analogy is the taxonomy of animals. There are animals, and then there are vertebrates and invertebrates, and there are mammals and reptiles and insects, and there are felines and canines, and there are tigers, and then there's this tiger that happens to have a litter of adorable tiger kittens, among which is a particular tiger kitten whose name is "Stripes". The structure (which is also called the *object hierarchy*) is like a family tree (or a root system), where everything is descended from a thing (an object) that came before it. The root object (with a unique number: #1) is unusual in that it has no parent.

Some objects are used so often that the system provides a way to refer to them by name instead of by object number, and we use a $-sign to designate those. Some especially common ones are `$thing`, `$container`, `$room`, and `$note`. But each of them also has a unique object number on the MOO (#5, #8, #3, and #9 respectively, on LambdaMOO).

[13] "MUD" has come to stand for "Multi-User Domain", though it's original meaning was "Multi-User Dungeon". MUDs have their roots in a role-playing game called "Dungeons and Dragons" and in some single-user computer games along the same lines that were popular in the 1970's and 1980's, notably *Adventure* and *Zork*.

When you create an object, you begin by specifying its parent and its name, for example:

```
@create $thing named "rock"
```

The system will respond with something like:

```
You now have rock with object number #1614 and parent
generic thing (#5).
```

And your rock will have all the attributes and characteristics of the generic thing that is its parent, until such time as you modify it, for example by giving it a description, or maybe by programming it to behave in some rock-like way.

You can see a list of all the objects you own by typing:

```
@audit me
```

You can see a list of objects Yib owns by typing `@audit Yib`. When you examine an object, you are told its number, among other things. You can check your parent object and its parent object(s) (i.e., its ancestors) by typing `@parents me`, and you can check the parents of any object by typing `@parents <object>`.

When must you refer to an object by its number, and when can you just use its name? In general, if you are holding an object, or are in the same room as an object, then you can refer to it by its name. If you are at some remove from an object (i.e. neither holding it and nor in the same room) then you generally have to refer to that object by its number in order for the system to know which object you mean. There are some exceptions: Objects that can be designated with the $-sign plus a name ($thing, etc.). Mailing lists, which begin with the asterisk symbol. Some commands let you specify a player by name even if you aren't in close proximity (`@who Klaatu`). And often you can specify a distant player by name with the tilde (~):

```
look ~yib
```

will yield the same result as:

```
look #58337
```

(if #58337 is Yib's object number).

On a MOO, everybody who's anybody, and everything that's anything, is an object.

See also `help objects`.

Moving Objects

Chapter 2 included a discussion of how to move (yourself) around the MOO. This section discusses moving other objects from one place to another.

A little about what it *means* to move an object: The system keeps meticulous track of every object's location. Specifically, every object has a property (a named piece of data) which stores that object's location in the form of an object number. An object in the LambdaMOO Living Room, say, would have the Living Room's object number in its `.location` property. (When naming a property, it is conventional to precede it with the "`.`" character.) Reciprocally, while that object is in the LambdaMOO Living Room, that object's number will appear in the Living Room's `.contents` property. So: every object stores its location, and every location stores a list of its contents. When an object is successfully moved in a MOO, three things happen: The object's `.location` property is changed to reflect the object's new location. The object is removed from the old location's list of contents. The object is added to the new location's list of contents.

The most straightforward way to move an object is to take it or drop it.

Suppose I own an object named `bright sparkly thing`, but leave it lying about in the driveway where anyone can find it. Shmool comes along and sees:

```
Driveway
A circular driveway, in front of LambdaHouse.  The
LambdaHouse front door is to the south.  The drive curves
away to the northeast and northwest; there is a spur to the
west, curving back around the house to the garage.
You see bright sparkly thing here.
```

Shmool types:

```
take bright sparkly thing
```

Shmool now has the bright sparkly thing in his *inventory* (the list of stuff he's carrying), and anyone looking at Shmool will see not only his description, but his inventory as well:

```
l Shmool

Shmool
A 3 1/2 foot tall squirrel, bald but for a 3 foot ponytail,
with large and luminous violet eyes.  A silver locket
dangles by a gossamer chain around his neck.
Carrying:
  bright sparkly thing
```

When Shmool took the bright sparkly thing, he changed the database, meaning the `.location` of the bright sparkly thing changed, the `.contents` of the driveway changed, and the `.contents` of Shmool changed. Note that if Shmool had merely emoted,`:takes bright sparkly thing`, on the other hand, while it might *appear* that he had taken it, it would in fact still be lying in the driveway.

Shmool might then type:

```
@go home
drop bright sparkly thing
```

to add it to his growing collection of stuff. The bright sparkly thing would be removed from Shmool's inventory and added to the contents of his room. If Shmool keeps this up, his room will become quite cluttered! He might be moved to create a treasure chest to put things in. Putting things into containers is another way to move things. Shmool would type:

```
put bright sparkly thing in treasure chest
```

and the bright sparkly thing would be moved again: Its new location would be the chest, and the contents of Shmool's room and his treasure chest would be changed appropriately, too.

Eventually I notice that my bright sparkly thing is missing. Where could it be? I look everywhere for it. Last seen in the driveway! I go to the driveway, but the bright sparkly thing is gone. Some rascal has taken it!

I might search high and low, and on a MOO as big as LambdaMOO, or even on a much smaller one, I might never find my object. If I cared to get my object back, this is an occasion where I might choose to break the VR and start working with object numbers. Specifically, I might want to teleport my object back to a location of my choosing.

Remember that if you aren't holding an object or in the same room with it (my predicament), then you must identify it by its number. For a while, I just ignored all those object numbers, and got along fine without them, but now I would *really* like to know the object number of my bright sparkly thing! There is a way, by using the `@audit` command. I type:

```
@audit
```

and the system prints out a list of objects (by number!) that I own, along with their size, name, and location. Now I have the information necessary to teleport my bright sparkly thing, either to myself or to my location or to a named object in my vicinity or to a location whose number I know (aha, those numbers, again), using the `@move` command. The syntax is:

```
@move <object> to <location>
```

Here are some of my choices, assuming that the object number of my bright sparkly thing is #4612 and that I am in a room with a walnut desk that I use in lieu of a treasure chest:

```
@move #4612 to here
@move #4612 to me
@move #4612 to a walnut desk
@move #4612 to #6193
```

(On LambdaMOO, the last example would move my bright sparkly thing back to the driveway.)

Getting Rid of Unwanted Objects

You may find yourself in the strange predicament that you are carrying something, or something is attached to you, or something is in a room you own which you would like to be rid of but which doesn't respond to the `@move` command. For this we have `@eject`. The syntax is:

```
@eject <item> from <location>
```

If you're holding it, then `<location>` would be `me`; if you want to eject a thing from a room you own, then `<location>` would be `here`.

Information is power. In this case, if you know the number of a thing, you can teleport that thing. There are some exceptions. (Aren't there always?) An owner can lock a thing in place, preventing people from taking it or otherwise moving it. An owner can put conditions on moving an object. The owner of a room can prevent an object from being dropped or moved there. (See locking, page 88) Lastly, on LambdaMOO, the owner of an object can let people borrow it, and have the housekeeper return it to a designated location when certain conditions are met.

Taking things that don't belong to you falls into a category I call, "risky behavior". It isn't strictly forbidden, and if you know how, it's easy to undo. But it also annoys some people, so think before you take.

Feature Objects

This section discusses Feature Objects, including an explanation of just what sort of objects they are and how they work.

When you are connected to a MOO, you type things in and read text that appears on your screen. The lines you type (except when you are being prompted for data, for example when you are working within an editor) are *commands*. When you type a command, you expect something to happen, either a change to the database (e.g. changing your location) or perhaps simply the display of some informational text from the database (e.g. looking at a room or a player). MOOs are user-extensible, which means that users can create objects and can define (i.e. program) commands associated with those objects, which you and others can then use. It is part of the server's job to consider the command you type in and divide it into its component parts (command name plus optional arguments or direct object, preposition and indirect object). The process of separating a command line into its component parts is called *parsing*, and the parser is the part of the server that does this. Once the command has been divided into its component parts, the server then tries to identify an object that defines the command (verb) that you want to run. It conducts its quest for an object defining the verb in a very particular order, as follows: (1) your player object or player class or any of its ancestors, (2) any of a player's feature objects, (3) the room you are in, (4) the direct object (if specified and identifiable), and (5) the indirect object (if present). If the parser finds that the command is valid (i.e. defined on one of player, a feature object, the room, the direct object or the indirect object), then the server tries to execute it. If the parser can't find any definition of the command on any of the objects it is supposed to consider, then the system prints the text, `I don't understand that.`

A feature object (commonly referred to as a/an FO) is an object that exists solely to serve as a repository for a set of commands that you might want to use, so that when you type the command, the system executes it rather than displaying everyone's favorite, `I don't understand that.` The beauty of feature objects is that for commands that lend themselves to this method implementation (some commands don't), anyone can use them *regardless of what player class e has selected.* This is why feature objects tend to have broad appeal.

Unlike objects that represent tangible things, such as chairs, candy, or rocks, you don't need to be holding an FO or in the same room with it in order to use it. Rather, one *adds* a feature to oneself, which is another way of saying that one adds its number to a list of feature objects one wishes to be able to use. To add a feature, you have to know its object number. To find out a feature's object number, you can try asking the person who just used it, or you can type:

```
@features for <so-and-so>
```

to see a list of that person's feature objects and their numbers. Then you can add it by typing:

```
@addfeature <object-number> to me
```

So, if you notice that mockturtle is a thoughtful guy, and in particular much of his text appears in typographical thought balloons instead of between double quote marks:

```
mockturtle . o O ( Can she read my toughts? )
```

You might say, "mockturtle, is that thought balloon verb on an FO?" And mockturtle might quickly type `@features for me` to jog his memory and then say, "Yes, it's the 'think' verb on the Thinking FO, #10392." And then you might type `@addfeature #10392` (and become more contemplative, yourself).

It's possible to use a feature object command quite often and forget which feature object it's actually on. Then when someone asks you what FO a verb is on, you might look at your list of features and still not know which one has the command in question. In such a case you might take advantage of a verb called `@find`. So instead of listing his feature objects, mockturtle might instead type:

```
@find :think
```

(Note that the colon is part of the command), and the server would print on his screen, `The verb :think is on Thinking Feature(#10392).`

Most feature objects have help text (e.g. `help #10392`) that list the commands they offer and explain briefly what they do and how to use them.

On LambdaMOO, there is an exhibit in the museum dedicated to feature objects, where you can read each one's help text and then pick and choose the ones you want. On other MOOs, you can type `@kids $feature` to see a list of all direct children of the generic feature object, and then read the help text for those feature objects that look like they might be of interest. It might be tempting to add all the feature objects you can find, but remember: When parsing a command, the server must consider all the verbs on you, your player class, its ancestors, *all your feature objects*, the room you're in, and possibly direct and indirect objects. The more feature objects you add to yourself, the longer this process takes. It's better to read the help text for various features and then add only those features that you think you'll actually use.

The sequence in which you add feature objects can matter. If two feature objects define the same command, the first one in your list of features is the one that will be executed. To change the order of your features, use the `@rmfeature` command to remove the one that comes first, then re-add it to move it to the end of the list.

Player Classes

A *player class* is an object that provides or expands a set of commands available for a player to use. To use a player class, a player changes eir parent to that player class object, thus inheriting all its properties and verbs and all its ancestors' properties and verbs.

Recall that every valid object (except #1, the root object) has another object as its parent. Players can change the parent of objects they own, *including themselves*, with the `@chparent` command:

```
@chparent <object> to <new-parent-object>
```

The list of an object's parents and ancestors is called its *parent hierarchy*. You can look at an object's parent hierarchy using the `@parents` command. The syntax is `@parents <object>`. To see a list of your own parent and ancestors, type:

```
@parents me
```

(If the system responds with `I don't understand that`, ask a wizard to make you a builder.)

Someone who has a fairly fancy player class (in this case on LambdaMOO) might have a parent hierarchy that looks like this:

```
Yib(#58337)
Sick's Sick Player Class(#49900)
Sick's Slightly Sick Player Class(#40099)
Sick's Sick of Spam player class(#59900)
Detailed Player Class(#6669)
Generic Super_Huh Player(#26026)
Politically Correct Featureful Player Class Created Because
Nobody Would @Copy Verbs To 8855(#33337)
Player Class hacked with eval that does substitutions and
assorted stuff(#8855)
Experimental Guinea Pig Class with Even More Features of
Dubious Utility(#5803)
Generic Player Class With Additional Features of Dubious
Utility(#7069)
generic programmer(#217)
generic builder(#630)
Generic LambdaMOO Citizen(#322)
Frand's player class(#3133)
Generic Mail Receiving Player(#100068)
generic player(#6)
Root Class(#1)
```

A brand new player on LambdaMOO would have a parent hierarchy that looks like this:

```
Bit_Blaster(#200119)
generic builder(#630)
Generic LambdaMOO Citizen(#322)
Frand's player class(#3133)
Generic Mail Receiving Player(#100068)
generic player(#6)
Root Class(#1)
```

A brand new player on another MOO would likely have a parent hierarchy that looks like this:

```
New_Grrl_On_The_Block(#1438)
generic builder(#4)
Frand's player class(#90)
Generic Mail Receiving Player(#40)
generic player(#6)
Root Class(#1)
```

(Note, however, that this may differ, as the wizards of each MOO can change the default starting player class for new players.)

In principle, it is vitally important to understand the workings of a player class you adopt, and to trust its owner and the owners of its ancestors. You can use the `examine` command to see an object's owner; many MOO's also provide a specific command to do this, e.g. `@owner`. At the very least, you should be aware that the owners of your player class and its ancestors are in a position to intercept and monitor your pages, intercept and read your private MOOmail, change your name and/or remove or change your aliases, and any number of other acts that you might normally expect to be your prerogative alone.

In practice, people often select a player class based on the recommendation of friends or experienced acquaintances, and trust the various player class authors by reputation. The purist in me would like to say that one should acquaint oneself with all the possible player class alternatives, read the help text and the verbs of each, understand what each does, and then select a player class based on the trustworthiness of the authors, and which is no fancier than one's particular needs at the moment. (The more elaborate the player class, the more quota your player-object takes up.) Note that it is trivial to change your parent to a fancier descendent of your current player class. It is less trivial to change to a different branch of the player class "tree", because doing so can mean having to give up messages or morphs (see glossary) in which one may have invested a fair bit of time and creativity.

There is generally sufficient social pressure on player class owner/authors not to spy or otherwise abuse their privileged position that it isn't a common problem. There was an incident several years ago on LambdaMOO in which the author of a popular player class was accused of intercepting pages which his girlfriend received from a perceived rival. There was an outcry when this was discovered, and the code was revised to the satisfaction of all concerned. In another instance, a player dared all comers to do their worst (the point being to demonstrate that in the face of any attack one could still MOO serenely without needing an arbitration system). The owner of one of this person's player class ancestors removed the challenger's name and aliases and changed them to something highly unflattering. The names and aliases were subsequently returned to their original owner. These instances are rare, but one should still be advised that adopting a player class entails a certain degree of calculated risk.

One should be especially wary if a player class has unreadable verbs. Type:

```
@display <player-class-object>:
```

to list the verbs. (The colon is part of the command.)

Knowing the commands available to you will give you better use of them. For more information about a particular command, try either of:

```
help me:<command>
help <command>
```

(Help text had not been standardized at the time these early player classes were written.)

Documentation for the oldest player classes that nearly every player on every MOO has in common is sometimes non-existent or difficult to find; here is a brief summary of what these basic player classes do. For a detailed explanation of any of these commands, see the command summary in Appendix A

- **Generic Player**: Provides the most basic set of commands, including `home`, `help`, `@describe`, `@gender`, `@quit`, `@password`, and `@mail-options`.
- **Generic Mail Receiving Player**: With the exception of `@mail-options` which is (incongruously) provided on the generic player, this player class provides all the commands for reading and sending MOOmail, including `@mail`, `@send`, `@read`, `@next`, and `@subscribe`.
- **Frand's Player Class**: Frand is one of LambdaMOO's oldest and most venerable and innovative players. So much so, in fact, that this player class is now included in LambdaCore itself, and has been integrated into JHCore. Many of the verbs on Frand's player class can be said to "break" or "transcend" the VR, including `@go`, `@join`, `@addroom`, `@rooms`, `@ways`, and `@refuse`.
- **Generic Builder**: This player class provides the minimum set of commands needed to add objects to the MOO's database, e.g. build rooms. They include `@create`, `@dig`, `@recycle`, `@chparent`, `@audit`, `@parents`, `@lock`, and `@contents`.
- **Generic Programmer**: In order to write programs on a MOO, you must have this player class in your ancestry and have gotten a programmer bit (the usual procedure for getting one is to ask a wizard). The programmer bit gives you the authority to program; the generic programmer player class gives you the necessary commands to do so. These commands include: `@property`, `@verb`, `eval`, `@program`, `@display` and `@dump`.

Setting Messages

```
Yib arrives in a shower of sparks.
```

Q: How does she *do* that?

A: With messages.

Recall that a *property* is a named piece of data associated with an object. A *message* is a special kind of property whose name ends with `_msg.` The purpose of messages is to give users a way to customize themselves and objects they own without having to learn how to program. Many kinds of objects have messages.

To see a list of the messages on an object, type:

```
@messages <object>
```

Player objects have a great many messages, and each player class in a player's ancestry typically adds a few more, so typing:

```
@messages me
```

will probably generate a long list! Let's look at two player messages:

```
@self_arrive #3133 is "%<teleports> in."
@oself_port #3133 is "%<teleports> out."
```

If you transport yourself within the MOO using either the `@go` or `@join` command, the system will process these messages (prepending your name (if it is absent) and conjugating the verb "teleport") and then display them to the appropriate viewers – at the location you are leaving and at your destination. To customize my departures and arrivals, then, I would change these messages. There are two syntaxes for doing so:

```
@set me.self_arrive_msg to "Yib arrives in a shower of sparks."
@set me.oself_port_msg to "Yib disappears in a sudden puff
of smoke."
```

or:

```
@self_arrive me is "Yib arrives in a shower of sparks."
@oself_port me is "Yib disappears in a sudden puff of
smoke."
```

Because *silent teleporting* (sneaking into a room without people knowing you are there) is frowned upon in most MOOs, the system displays your name at the beginning of these messages if it isn't otherwise present. Furthermore, because some players may have their gender set to plural, the default messages use the syntax `%<teleports>` to signal that "teleports" is a verb that should be conjugated.

The messages on different player classes (including the ancestors of *your* player class!) were created by different people at different times, and many of them were created before some now-common conventions were established. Unfortunately, many of them aren't documented, or are documented poorly. Historically, people figured out and set most of their own messages by listing them (`@messages me`), looking at the combination of their names and content, and (often) trying them out with a friend.

Here, then, is a list of all the messages that you are likely have defined on your player object as a new arrival on a MOO that uses LambdaCore, along with a few comments about some of the messages' idiosyncrasies. They are presented as if you had typed `@messages me`, and the syntax for setting them is the same as that displayed by the system when it lists them.

```
@more me is "*** MORE ***  %n lines left.  Do @more
[rest|flush] for more."
```

`@more` is the message displayed if you have used the `@pagelen` command to set a fixed page length. The syntax `[rest|flush]` means that you have the option of typing `@more rest` or `@more flush` to print all of the remaining output, rather than just one additional page. Type `@more` by itself to see just the next page.

```
@page_absent me is "%N is not currently logged in."
```

`@page_absent` is the message that someone sees if e pages you when you are logged off. In this message, your name will be substituted for `%N`.

```
@page_origin me is "You sense that %N is looking for you in
%l."
```

`@page_origin` is the message that someone sees as when you page em, before the actual content of your paged text is displayed. Your name is substituted for `%N` and the name of your location is substituted for `%l`.

```
@page_echo me is "Your message has been sent."
```

`@page_echo` is the acknowledgement message that someone sees when e pages you.

```
@join me is "You join %n."
```

This message is displayed to you when you use the `@join` command to teleport to another player's location. The name of the player you are joining is substituted for `%N`.

```
@object_port me is "teleports you."
```

This message is transmitted to an object it when you teleport (i.e. `@move`) it. (See the section on moving objects, page 38.) Your name (followed by a space) is automatically added at the beginning.

```
@victim_port me is "teleports you."
```

This message is displayed to a player if you teleport em somewhere. Your name (followed by a space) is automatically added at the beginning.

```
@thing_arrive me is "%T teleports %n in."
```

This message is displayed to your location when you teleport an object either to yourself or to the room you are in. Your name is substituted for `%T` and the name of the object being teleported is substituted for `%n`.

```
@othing_port me is "%T teleports %n out."
```

This message is displayed to the room a thing is in (if it is in a room) when you teleport that thing to a different location. Your name is substituted for `%T` and the name of the object being teleported is substituted for `%n`.

```
@thing_port me is "You teleport %n."
```

This message is displayed to you when you teleport an object. The name of the object you are teleporting is substituted for `%n`.

```
@player_arrive me is "%T teleports %n in."
```

If you teleport (i.e. `@move`) a player, this message is displayed to the room to which that player is moved. Your named is substituted for `%T` and the name of the player being teleported is substituted for `%n`.

```
@oplayer_port me is "%T teleports %n out."
```

If you teleport (i.e. `@move`) a player, this message is displayed to the room from which that player is removed. Your named is substituted for `%T` and the name of the player being teleported is substituted for `%n`.

```
@player_port me is "You teleport %n."
```

This message is displayed to you when you teleport (i.e. `@move`) a player. The name of the player you are moving is substituted for `%n`.

```
@self_arrive me is "%<teleports> in."
```

This message is displayed to your destination when you teleport somewhere (e.g. with `@join` or `@go`). Notice that it is not a complete sentence. Your name is automatically prepended to the beginning of the message if it doesn't appear somewhere else within it. Setting this message to the empty string (`""`) will result in a silent arrival, which is contrary to good manners, and in some places might be construed as a form of spying. The construct `%<teleports>` causes the verb "teleport" to be conjugated according to your `.gender` property.

```
@oself_port me is "%<teleports> out."
```

This message is displayed to your point of departure when you teleport out. Again, your name is prepended if it doesn't appear somewhere else in the message. Setting this message to the empty string will result in a silent departure. While not considered as bad as a silent arrival, it can be unsettling to others in the room not to know that you have left. . The construct `%<teleports>` causes the verb "teleport" to be conjugated according to your `.gender` property.

```
@self_port me is ""
```

This message is displayed to you when you teleport somewhere. It is blank by default.

```
@page_refused me is "%N refuses your page."
```

This message is displayed to someone if e tries to page you but you have refused pages from em. (See page 30) Your name is substituted for `%N`.

```
    @whisper_refused me is "%N refuses your whisper."
```

This message is displayed to someone if e tries to whisper to you and you have refused whispers from em. Your name is substituted for `%N`.

```
    @mail_refused me is "%N refuses your mail."
```

This message is displayed to someone if e tries to send you mail and you have refused mail from em. Your name is substituted for `%N`.