Chapter 5 – Extending the Virtual Reality: Building

Overview

This section details the various ways to create rooms and other objects. It amplifies what you already leaned about `@dig`. On nearly all MOOs, players start out as members of the “generic builder” player class, which means that the building commands are available to them.

Between the basics of communication, moving around, and interacting with various objects on the one hand, and the intricacies of programming on the other, is building. Building is the business of creating new objects on a MOO and modifying them in certain limited ways. When you create an object, you first specify what kind of object it is to be (a room or a container or a note, for example), and the object’s name, with optional aliases. Then, typically, you describe your object, maybe set some of its messages (see page 46), and the object is then ready to use. Some fancier kinds of objects let you specify some additional information which might also govern how the object behaves. An example of this might be where an object should go if it needs to be “sent home”.

@create

```
    @create <generic> named "<name>"
```

or:

```
    @create <generic> named
    "<name>","<alias1>","<alias2>",…,"<last alias>"
```

`@create and @dig` are the two quintessential commands of building. Each creates a new object where there was none before.

There are a couple of ways to specify the kind of thing (i.e. the *parent* or *generic*) you want to create. As always, you can specify the object by its object number, if you know it. If the generic you want to create a copy of is in your vicinity (i.e. you are either holding it or in the same room with it), then you can specify it by name. A few objects are used so commonly that we can refer to them even if they are not nearby, using the “$” sign. The generic thing is called `$thing`. The generic container is called `$container`. The generic note is called `$note`. In this way, you can create things and containers and notes without having to know the generics’ object numbers or have them in your vicinity. Let’s look at an example of creating a very simple object, a paper weight:

```
    @create $thing named "paper weight","paperweight","pw"
```

The system will print out something like:

```
You now have paper weight (aka paperweight and pw) with
object number #63555 and parent generic thing (#5).
```

The actual number of your paper weight will be different. Do you have to remember this number? Yes and no. As long as the paper weight is either in the same room that you are in or is in your inventory (i.e. you are holding it), then you can refer to it by name. And when you first create something, you are holding it. That's one reason why we use `@dig` (described further on) instead of `@create` for rooms and connecting exits: so that you won't be holding these things when they are created. There's nothing actually *wrong* with holding rooms or exits, but it doesn't make sense and serves no practical purpose.

@describe

The next logical thing to do is to describe your paper weight:

```
@describe pw as "An ovoid paper weight made of onyx.  Though
perfectly smooth, it has the curious property that it gives
no reflection, almost as if it were an oddly-shaped black
hole.  It does have the expected flat spot on the bottom."
```

There isn't much you can do with a $thing. You can hand it to someone. You can drop it. You can take it. That's about it. To be sure, type:

```
examine paper weight
```

What you *can* do is change the text that you and/or others see when you hand it to someone or drop it or take it. (Note that you don't have to be a builder to set messages on objects you own; I review it here because it's the logical next thing to do when crafting an object.) This is done through messages, and changing the messages on an object is a quick and easy way to give it a bit of character. First, you might want to list the existing messages:

@messages

```
@messages paper weight
```

The system will respond with the following list:

```
@drop_failed paper weight is "You can't seem to drop %t
here."
@drop_succeeded paper weight is "You drop %t."
@odrop_failed paper weight is "tries to drop %t but fails!"
@odrop_succeeded paper weight is "drops %t."
@otake_succeeded paper weight is "picks up %t."
@otake_failed paper weight isn't set.
@take_succeeded paper weight is "You take %t."
@take_failed paper weight is "You can't pick that up."
```

Because your object is a child of `$thing`, it has inherited all its properties, including all the messages that `$thing` has.

The purpose of most of these messages should be pretty clear from the combination of their names and content. By convention, messages beginning with “o” are told to others, while messages not beginning with “o” are told to the player initiating the relevant action. The `%t` (think “this”) in each message substitutes the actual name of the object. In general, most generics have their messages set so that they make sense without any customization, but let’s change a few of these to demonstrate the method. The syntax for setting messages is the same as the way the messages are printed out above. Here are a couple of examples:

```
@drop_succeeded pw is "You drop %t.  It lands with a thud,
then rolls a short distance before coming to a stop."

@odrop_succeeded pw is "drops %t.  It lands with a thud,
then rolls a short distance before coming to a stop."
```

@recycle

Should you decide that you no longer want this object, you can get rid of it by typing:

```
@recycle paper weight
```

and the system will respond with a line like:

```
paper weight (#63555) recycled.
```

You can only recycle objects that you own. If the object you wish to recycle is not in your vicinity, then you can recycle it by object number instead of by name. It’s good to recycle objects that you don’t use, as this conserves system resources.

Generic Objects

$things are useful as stage props. If you want to be seen carrying a feather duster, for example, but don’t need to do any actual dusting, `@create $thing named "feather duster"` will probably suit your purposes adequately. Other generics, however, are capable of doing more. A good example of this is the generic container, abbreviated `$container`. To make one, type (for example):

```
@create $container named "leather pouch","pouch","lp"
@describe lp as "A simple-looking leather pouch, of
remarkable capacity."
```

You can type `examine pouch` to see what actions you can actually do with it, and `@messages pouch` to see if you want to change any of the associated messages. Then if you want to you can put your paper weight in the pouch.

Normally, one would (and should) investigate a generic before making an instance of it. People tend to practice varying levels of due diligence in this regard, but the appropriate steps (the sequence isn't all that important) are as follows:

```
examine <generic>
help <generic>
@parents <generic>
```

Then repeat the above for each listed parent until you come to an object you already know about.

Generics can be extremely complex. By using `@create` to make an instance (think "clone") of an existing generic, you get all its functionality without having to do any programming. Building things based on existing generics also takes up less space in the database than programming a new object from scratch, which is desirable, too.

So then the question arises, "How does one know what generics there are to make kids of?" Some of the basics are `$thing`, `$container` (you can open it, close it, put things into it and take things from it), `$note` (you can edit its text and others can read it), `$letter` (like a `$note`, but a designated recipient can burn it when e has finished reading) and `$mail_recipient` (a MOO mailing list). These generics are part of the LambdaCore, and are included in every database based on it. Players who are programmers can create additional generics. These usually can't be referenced by a name beginning with a "$"; they have to be referenced by object number.

@parents

One way to learn about existing generics is to explore the MOO, examining many objects, and when you find one that interests you, type:

```
@parents <object>
```

This command shows an object's ancestry, and after the `examine` command it is one of the most basic tools at your disposal to investigate an existing object. Suppose, for example, you took an interest in a hot air balloon that you came across in your explorations. You could type:

```
@parents Royal Blue Balloon

Royal Blue Balloon(#68806)   Generic Hot Air Balloon(#66549)
Generic Aircraft(#42055)   Generic Magnetic Portable Secure
Seated Integrated Detail Room(#58237)   Generic Portable
Secure Seated Integrated Detail Room with Sensible
Locking(#17524)   Generic Portable Secure Seated Integrated
Detail Room(#36643)   Generic Secure Seated Integrating
Detailed Room(#9805) Area/Seat-Conscious Room(#5531)
Generic Secure Integrating Detailed Room (without
seats)(#156)   Integrating Detail Room with Features(#21311)
Integrating Detail Room with Exit-Verb Matching(#8801)
Integrating Detail Room Mark III(#17755)   Modified Detail
Room(#11825)   Frand's generic detailed room(#6464)   Self-
Cleaning Room(#27777)   generic room(#3)   Root Class(#1)
```

Whew, that's a lot! (You'll find that some objects have very long pedigrees.) You could then read the help text for the generic hot air balloon by typing `help #66549` and make an informed decision as to whether you actually wanted to create one for yourself. Or you might decide that instead of a hot air balloon, a flying carpet is more to your taste, in which case the generic aircraft might be a better choice.

LambdaMOO also has a museum, as do some other MOOs. This is a room or set of rooms whose purpose is to provide information about various generic objects that are available, and it is a valuable resource.

Not every item is available as a generic; sometimes a programmer might make an object but want to limit it to being one-of-a-kind. Before people can make their own copies of an object the programmer of that object must make it *fertile*. (The command is `@chmod <object> +f`, but that's properly in a section about programming rather than building. It's included here for completeness.) If an object is not fertile, you can't make a copy of it, even if it has the word "generic" in its name. You are out of luck unless you can persuade the owner to make it fertile.

@kids

This command is the opposite of `@parents`. You use it to list the children of a given object. Note that it does not list kids of kids. Different MOOs have different commands to list all descendents with one single command, but you can always use `@kids` sequentially to explore various branches of the *object hierarchy*.

@audit

It is a common predicament to create an object (or a room, see `@dig`, below), then misplace it and not be able to find it or use it because you've forgotten its object number. For this we have the `@audit` command. There are two forms:

```
@audit
@audit <player>
```

The first audits yourself. It prints out a list of all the objects that you own, along with their object numbers and sizes. The second form prints the same information, except that it lists another player's objects instead of your own. Other players can `@audit` you and see a list of things that you own. Suppose I mislaid my paper weight, and typed `@audit` to find it. I might see something like this:

```
Objects owned by Yib (from #0 to #118569):
 54K  #58337 Yib                        [Yib's Study]
  4K  #23920 Yib's Study
 <1K  #57744 a walnut desk              [Yib's Study]
  2K  #71354 white dendrobium orchids   [Yib's Study]
  3K  #32504 a linen handkerchief       [Yib]
 <1K  #35487 west                       Yib's Study->*Library Turret
  3K #107539 east                       *Library Turret->Yib's Study
  1K  #71176 a few lucky Bits           [Yib]
  1K   #4612 bright sparkly thing       [a walnut desk]
  3K #101204 a lady's pocket watch     *[Boo]
 <1K  #63555 paper weight              *[Under the Couch Cushions]
-- 11 objects.  Total bytes: 74,927.------------------------------
```

The numbers in the first column are the sizes of the objects in kilobytes of quota. The second column is the object number and name of each thing I own, including myself. The third column tells the name of each object's location (in square brackets); an asterisk(*) indicates that I don't own the indicated location. If the object is an exit, then the third column shows the names of the two rooms each exit connects together, instead. Here you can see that my paper weight has somehow found its way to a place called "Under the Couch Cushions". I can retrieve it easily by typing:

```
@move #63555 to me
```

@dig (rooms)

As mentioned previously, when you `@create` something, its initial `.location` is you – that is to say, you will be holding it right after creating it. In the case of a room, this is awkward, because holding a room doesn't make sense (unless perhaps it's a portable room) and also, you can't teleport into or otherwise enter a room if you are holding it, because it would violate the *containment hierarchy.* (`A` can't be located inside `B` if `B` is located inside `A`). So, for rooms, we use the `@dig` command instead. We also use `@dig` to create the exits which connect two rooms, because it conveniently automates the administrative work of setting the exit's source and destination. Here are some examples of each form:

```
@dig Home Sweet Home
```

The system will respond with, `Home Sweet Home (#113415) created`, thus informing me of the object number of my new room. (Note that the object number of your room will be different.) Now I can teleport there, and describe it. If I ever forget its number, I can always `@audit` myself to find out again.

The location of this new room is `#-1  <$nothing>`. You can think of it floating free in the ether. Until you connect it to another room with exits (see below), you can only get there by teleporting. There is nothing wrong with this, by the way – many rooms are unconnected, and that's just fine.

@dig (exits)

An exit is a special kind of object that connects one room to another. It's special in that it isn't located *in* either of the rooms it connects, but can be referenced by name in the room that is its `.source`. The other room is the exit's `.dest` (think "destination"). To have a two-way connection between a pair of rooms, you have to have two exit objects, one for each direction.

Digging an exit starts in the source room. You can dig exits one at a time, or two at a time (one for each direction), you can specify aliases at the time you `@dig` or add them later, and you can dig to an existing room or create a new room simultaneously. Here are some examples:

```
    @dig "east" to #115
```
would dig an exit named "east" to the room with the object number #115.

```
    @dig "east","e","out" to #115
```
would dig an exit with the aliases "east", "e", and "out" to the room with object number #115.

```
    @dig "east","e","out"|"west","w","in" to #115
```
would dig two exits in opposite directions, connecting room #115 to the room you were in when you typed the `@dig` command. The vertical bar (`|`) character separates the aliases of the exit going *to* the room from the aliases of the exits leading back *from* the room.

```
    @dig "east","e","out" to The Back Porch
```
would create a new room named "`The Back Porch`", and simultaneously create an exit, `east`, from the room where you were to the new room.

```
    @dig "east","e","out"|"west","w","in" to The Back Porch
```
would create a new room named "The Back Porch" and exits in both directions connecting the porch to the room you were in when you typed the `@dig` command.

The system will print a line informing you of the object numbers of the new exits (and, if applicable, the new room) that you've created with the `@dig` command. As always, you can also get the object numbers of these newly-created objects using the `@audit` command.

To make your building richer, it's good to describe your exits and set their messages. This is addressed in the section on room integration and exit messages, starting on page 94.

Exit objects must designate the rooms they lead from and to; this is normally done automatically as a side effect of the `@dig` command if you own both rooms. If you don't own the room the exit leads from, the owner of the source room should use the following command to attach it:

```
    @add-exit <exit-object>
```

And if you don't own the room that is the exit object's destination, then the owner of the destination room must use the following command to complete the connection:

```
@add-entrance <exit-object>
```

Exit attachment matters more for the source room, since without the attachment, nobody can use the exit. If the exit leads *to* a room that has been set not to allow teleportation via the `.free_entry` property, the exit can't work unless it's attached. The principle, here, is that if a room permits a person to teleport in, then the message generated by an exit is no worse than a teleport message. On the other hand, if the owner of the room won't let you teleport in, then you can't dig your own exit to the place, either. Or rather you can `@dig` the exit, but you can't use it unless the destination's owner "blesses it" with the `@add-entrance` command.

@chparent

The `@chparent` command is typically used to change the parent of an object from its current parent to another generic, usually (though not necessarily) an object of a similar type. One might `@chparent` a room, for example, to a fancier -- or merely different -- room generic. Likewise one might change the parent of an exit to a transparent exit, or a generic gate, and one might `@chparent` oneself to a different player class. For example:

```
@chparent me to #191
```

or:

```
@chparent here to #9805
```

@recreate

The `@recreate` command is basically a combination of `@create` and `@chparent`. When you use `@chparent`, any properties and messages that you've set that are in common with the new parent keep the values you've set them to. If you really want to start over from scratch, but keep the same object number, then `@recreate` is the command to use. The syntax is:

```
@recreate <object> as <parent> named <new name>
```

@set

Some objects permit some customization by letting you set the value of one or more properties which in turn affect the object's behavior. You can, for example, modify the way a room's contents are displayed by setting its `.ctype` property to different values. To the best of my knowledge, there isn't any documentation about the various possible values of `.ctype`, so I present a rather long illustration here.

```
       @set <room>.ctype to 0
```
will list out the room's contents, one item per line, in the order in which the items entered the room:

```
       The Front Veranda
       A gloriously spacious covered veranda, painted all in white.
       To the east, a large door leads into the mansion.  Wide
       steps lead west and down to the front lawn.
       Contents:
         YibCo Muffle-Matic Soundproof Energy Field
         Rocking Chair
         a small wicker basket
         a white-washed wooden porch swing
         Yib
```

```
       @set <room>.cytpe to 1
```
will put all non-player objects into separate sentences of the form, "You see <item> here," on separate lines, and all players into separate sentences of the form "<So-and-so> is here," on separate lines. The order will be the order in which items and players entered the room:

```
       The Front Veranda
       A gloriously spacious covered veranda, painted all in white.
       To the east, a large door leads into the mansion.  Wide
       steps lead west and down to the front lawn.
       You see YibCo Muffle-Matic Soundproof Energy Field here.
       You see Rocking Chair here.
       You see a small wicker basket here.
       You see a white-washed wooden porch swing here.
       Yib is here.
```

```
       @set <room>.ctype to 2
```
will list all the contents in a single sentence:

```
       The Front Veranda
       A gloriously spacious covered veranda, painted all in white.
       To the east, a large door leads into the mansion.  Wide
       steps lead west and down to the front lawn.
       You see YibCo Muffle-Matic Soundproof Energy Field, Rocking
       Chair, a small wicker basket, a white-washed wooden porch
       swing, and Yib here.
```

```
       @set <room>.ctype to 3
```
will list items in one sentence, and players separately, in another:

```
       The Front Veranda
       A gloriously spacious covered veranda, painted all in white.
```

```
To the east, a large door leads into the mansion.  Wide
steps lead west and down to the front lawn.
You see YibCo Muffle-Matic Soundproof Energy Field, Rocking
Chair, a small wicker basket, and a white-washed wooden
porch swing here.
Yib is here.
```

If you set a room's `.ctype` to anything else, the contents won't display at all, *unless* you have changed its parent to a room generic that supports additional `.ctypes`. (Ideally, the room's help text will mention this. To read a room's help text, type `help here` while in the room or `help <room object number>` if you are not in the room.)

Using @set with Messages

A message is a property on an object that ends with "_msg". These properties are special in that they can be set as described in the larger section on messages (see page 46), and that is the way it is usually done. But message properties can also be set and/or changed using the `@set` command, just like any other property of an object. If you had a paper weight with the alias "pw", then the following two commands would have an identical result:

```
@drop_succeeded pw is "You drop %t.  It lands with a thud,
then rolls a short distance before coming to a stop."

@set pw.drop_succeeded_msg to "You drop %t.  It lands with a
thud, then rolls a short distance before coming to a stop."
```

@contents

One advantage of being a `$builder` is that you don't have to depend on a room's description (or any object's description) to find out what its contents are. You can type:

```
@contents <object>
```

and get a list of its contents by name and number. Like all of the commands presented in this segment, it is a meta-VR command, which crosses the boundary of a MOO's theme into its underlying structure. The trade-off is some of the VR charm for increased informational accuracy.

@lock

As a builder, you can control some of the ways your objects are used. The easiest way is with the `@lock` command. `@lock` works differently with different kinds of objects, and that can make it seem a bit tricky, but it's easy once you get the hang of it. Briefly, if you lock a room, that governs what can and cannot enter it. If you lock a thing or a container, that governs locations to which it can and cannot be moved. And if you lock an exit, that governs who (or what) can and cannot pass through it. The syntax of the command is:

```
@lock <object> with <key>
```

To use `@lock` effectively, you need to understand the concept of a key. A key is a string of text that represents objects and ways that they can be combined. Briefly, "&&" means "and", "`||`" means "or", and "`!`" means "not". These can be combined in various ways. For example, if my object number is #97, and Ostrich's object number is #891, then "`#97 && #891`" means "Yib and Ostrich", "`#97 || #891`" means "Yib or Ostrich", and "`!#97`" means "not Yib". You can combine any number of objects in any number of ways. Use parentheses to clarify complicated expressions. (See also `help locking` and `help keys` online.)

The reverse of locking an item is:

```
@unlock <item>
```

Containers offer the additional option of:

```
@lock-for-open <container> with <key>
```

and:

```
@unlock-for-open <container>
```

This governs who can open a container, as opposed to who may take a container or where a container may be dropped.

@build-options

There is an interface called an *options package* that lets you customize the way some of the building commands work. To list your current option settings type:

```
@build-options
```

Most MOOs support four builder options; this section explains how to set and clear each of them, and what each of them means.

```
@build-option dig_room=<room-generic>
```

When you `@dig` a room, you are creating a child of a particular room. In most cases, this is `$room`, which is the generic room provided by the MOO. After digging a room, you may wish to use the `@chparent` command to select a different room-generic as the room's parent. If you have a favorite room generic and want all the rooms you dig to have that generic as a parent, you can specify that with this option. To clear this option and make it so that all rooms that you subsequently `@dig` use the system default, type `@build-option -dig_room`.

```
@build-option dig_exit=<exit-generic>
```

As with rooms, when you `@dig` an exit, its parent is set to a particular generic exit, usually `$exit`. If you wish to specify a different generic exit as your default, you can use the `dig_exit` builder option to specify a generic, and all exits you `@dig` after that will be kids of the generic you specified. To clear this option and make it so that all exits you subsequently `@dig` use the system default, type: `@build-option -dig_exit`

```
@build-option create_flags=<flags>
```

This option governs the permission settings that will be associated with every object you create. An object can be readable by others or not, writable by others or not, and fertile or not. Readable means that others can list the properties on your object. Writable means that others may add or remove properties and/or verbs from your object. CAUTION! It is almost never a good idea to set an object to be writable. Better to use available facilities or get a wizard to change ownership of an object to a different person if you want to let someone else assist you with your building. Fertile means that other people may make kids of your object. The value for `<flags>` in this builder option can be any substring of `"rwf"`, or it can be the empty string `""`).

```
@build-option -bi_create
@build-option +bi_create
```

When you create a new object, the system will either re-use a previously-recycled object, or it will create a new object with a higher object number than all previously-created objects. It is better for the database if you use recycled objects, which is the “`-bi_create`” option.

@quota

Quota is the term we use to measure the amount of space that objects take up in the computer’s memory. Building things takes up space in the database, and players are usually granted a fixed amount of quota to start with. In LambdaMOO’s early days, players were allotted a fixed number of objects that they could create. An unforeseen consequence of this was that people programmed fancier and fancier objects which took up more and more space, culminating in the generic multi-room, which was a room that simulated many rooms but which was, in fact, still a single object. LambdaMOO then converted to what is called “byte-based quota”. A player may create as many objects as e wishes, except that the total size of all the objects may not exceed a specified limit. (If it does, then the player can’t create new objects or add properties to existing objects.) The command:

```
@quota
```

will display how much quota you have used up with the objects you have created, and how much you still have available for creating new objects. You can also type:

```
@quota <player>
```

to see how much quota someone else has available, and how much e has used up.

Different MOOs will have different policies regarding whether they use object-based quota or byte-based quota, how much quota players are allotted when they first register, and how to get more quota. LambdaMOO has an elected Architecture Review Board (ARB) which reviews quota requests against certain criteria; in addition, LambdaMOO players can transfer quota directly to one another. On other MOOs, wizards will set quota policy.

Trimming Down Your (Quota) Size

@measure

The amount of quota an object takes up can change. Consider a note that has only a few lines of text in it. Then its owner edits it to be an extremely long note. Now it takes up more quota. The system has a single object-measurement task, and individual objects are generally measured only once every few days. Thus, the information provided by `@quota` may not be up-to-date. This shouldn't matter unless you are over quota and are trying to "slim down", as we say, perhaps in order to be able to create another object. In addition, you are only permitted to have a certain maximum number of unmeasured objects at a time (ten, on LambdaMOO), and after that you may not create more objects until the new objects have been measured. The `@measure` command, in its several variations, is provided so that players don't have to wait for the automatic measurement task to run if they need or want to have an object measured sooner than that. The tradeoff is that measuring things takes up computational resources, contributing to lag, and players are asked to use this command sparingly.

```
@measure object <object>
```

This command is used to measure a single object at a time, in lieu of waiting for the background measurement task to get to it.

The first step for trimming down is to use the `@rmm` command to remove any MOOmail messages that you don't need to keep. (You can use the `@netforward` command to forward messages to your registration email address before removing them, too.) These removed messages are not entirely gone, yet. The next step is to expunge the removed messages. You can do that in either of the following ways: `@renumber me` renumbers all your MOOmail messages and expunges deleted messages. `@unrmm expunge on me` expunges removed messages without renumbering them. Last, type `@measure object me` and `@measure summary`, to measure and record your new (smaller) size.

```
@measure summary
@measure summary <player>
```

The `@quota` command does not itself measure objects. Rather, it prints out summary information that was computed at an earlier time. `@measure summary` will tally up the current total of your object sizes for reporting by the `@quota` command.

```
@measure new
@measure new [<player>]
```

In a MOO that uses byte-based quota, you can only have a fixed number of unmeasured objects. After that, you can't create any new objects until the current ones have been measured. This can be a problem if you need to create a large number of small objects. They don't take up more than your allotment of quota, but still you can't `@create` more until they've been measured. The symptom of this particular problem is an error message that reads, "`Resource limit exceeded`." `@measure new` alleviates this. Once the new objects have been measured, you can go on to create more if you want to. If you are assisting someone with this dilemma, you can type `@measure new <person>` to measure that person's new objects instead of your own.

```
@measure recent [<number of days>] [<player>]
```

This measures those objects (yours or the specified player's) that haven't been measured either in the specified number of days, or, if no number of days is specified, the ordinary cycle of the measurement task. If no player is specified, then it measures your objects.

```
@measure breakdown <object>
```

If you just can't figure out why an object is SO BIG, the `@measure breakdown` command will print a list of how much space each property and verb takes up, and, hopefully, provide you with some clues. You can optionally have the output sent to you via MOOmail, but be aware that the message itself takes up quota.

@newmessage, @unmessage

In general, only programmers can add properties to and remove properties from an object, but builders can add and remove message properties. This is a pretty obscure aspect of building, and I'm only going to treat it briefly. There are a very few occasions where different things will happen depending on the presence or absence of a certain message properties. Some room descriptions will incorporate an object's `.look_msg` property, if present. If an object has a `.carried_msg` property, some player classes will integrate the message into a player's description.

A builder who does not have programming privileges can still add a message property as follows:

```
@newmessage <message-name> [<message>] [on <object>]
```

and remove a message from an object if it is no longer wanted or needed:

```
@unmessage <message-name> [from <object>]
```

@check-chparent

This command would be useful if you had added a message to an object, then tried to change the parent of that object to a generic that also had the same message. An object can define a property or inherit it, but not both. Programmers probably find more use for this verb than builders.

Creating a Mailing List

Building a mailing list is a popular thing to do, but requires extra steps so others besides yourself can use your mailing list, too. First, create the list, using the generic mail recipient as a parent:

```
@create $mail_recipient named <new list name>
```

Next, give your list a description, explaining what its intended topic is:

```
@describe <your list> as "<description>"
```

Then, to make it a public list that anyone can read and to which anyone can post, do:

```
@set <your list>.readers to 1
```

Last, to make your mailing list publicly available so that people can subscribe to it, do:

```
@move <your list> to $mail_agent
```

Note that $mail_agent won’t accept a list that lacks a description. See also `help $mail_recipient`.

Room Integration and Exit Messages

Integrating objects into a room’s description, adding messages to exits, and (to a lesser extent) describing exits can enrich a user’s VR experience at the expense of relatively little effort on the part of a builder. This extended example illustrates each of these techniques.

Suppose you have a front room and a porch, situated east-west relative to each other, and a screen door in-between:

```
    +-----------+
    |           |---------+
    |   Front   |         |
    |   Room
    |           |         |
    |           |         |
+---+---  -+----+  Porch  |
|          |              |
| Kitchen  |              |
+----------+--------------+
```

```
The Front Room
You are in the front room of a guest cottage.  There are a
few chairs and a braided rug.  A small kitchen is to the
south.  There is a screen door to the east.


Porch
You are on a breezy, screened-in porch.  A rocking chair and
a porch swing invite you to stay and relax for a while.  A
screen door leads west into the cottage, steps lead down to
the lawn.
```

Here are the steps you would follow to make an integrating room with exit messages:

1. Start by making the rooms’ descriptions clearly mention the obvious exits, so that people don’t have to guess or use meta-VR commands such as `@ways`, since this is intended to be a welcoming place for people to visit, and not a puzzle.

2. Describe the exits. This means describing what someone would see if they looked in the direction of the exit, for example, `look east`.

Stand in the front room and type:

```
@describe east as "You see a weathered but sturdy screen
door, held closed by a spring.  The top screen has a small
tear in the lower left-hand corner."
```

Stand on the porch and type:

```
@describe west as "You see a weathered but sturdy screen
door.  The handle and hinges are rusty but serviceable.  The
top screen has a small tear in the lower right-hand corner."
```

Notice that in writing these descriptions, I have implicitly decided that the door opens outwards onto the porch -- that's the side that the hinges are on. We can use this detail later to intensify the "VR feel" of things. The descriptions don't have to be elaborate, but it's nice if they add some new information to what's already there in the rooms' descriptions.

3. (Optional, but nice) Use `look_msg`s for the exits' descriptions instead of describing them in the room's description proper. (Yes, this contradicts step 1.) The reason for doing this is so that the exits will consistently be mentioned at the end of the description, no matter how many other objects' `.look_msgs` are included.

Some rooms can integrate objects and exits into their descriptions.[15] By convention (on LambdaMOO and YibMOO at least), an integrating room checks to see which objects and exits have a `.look_msg` property and/or a `:look_msg` verb, and, if so, incorporates those messages into the description instead of baldly listing the them in the room's contents afterwards (in the case of objects).

Why bother with this? Lets start by redescribing our Front Room, which had exits leading southwest to the kitchen, stairs leading up, and our east exit onto the porch. And let's put in a fireplace object, so we can see how the `.look_msg` properties interact. Starting with no `look_msg` properties on any objects or exits:

```
The Front Room
You are in the front room of a guest cottage.  There are a
few chairs and a braided rug.
You see fireplace here.

@prop south.look_msg "A small kitchen is to the south"
@prop east.look_msg "There is a screen door to the east."
```

Now if we were to look at the room, we'd see this:

```
The Front Room
You are in the front room of a guest cottage.  There are a
few chairs and a braided rug.  A small kitchen is to the
south.  There is a screen door to the east.
You see fireplace here.
```

[15] One integrating room generic on LambdaMOO is #17755 (Integrating Detail Room Mark III). See also #9805. On YibMOO, you can use the YibCo(tm) Multi-Media Modular Room (#237) in conjunction with the Integrating Description Module (#259).

This is strikingly like what we had before, but watch this. Now we'll put a `.look_msg` on the fireplace, too:

```
@prop fireplace.look_msg "Against the west wall is a large
stone fireplace."
```

Now the description becomes:

```
The Front Room
You are in the front room of a guest cottage.  There are a
few chairs and a braided rug.  Against the west wall is a
large stone fireplace.  A small kitchen is to the south.
There is a screen door to the east.
```

We could add a painting (assuming you have an object that is a painting):

```
@prop painting.look_msg "A portrait of someone, vaguely
familiar, hangs on the north wall."
```

```
The Front Room
You are in the front room of a guest cottage.  There are a
few chairs and a braided rug.  Against the west wall is a
large stone fireplace.  A portrait of someone, vaguely
familiar, hangs on the north wall.  A small kitchen is to
the south.  There is a screen door to the east.
```

The beauty of it is that you can integrate any number of objects into the middle of the description, and still have the exits described at the end, where they are easy to find.

4. Give the exits messages. Exit messages govern what the player sees and what others see when a person goes through the exit. There are six to set (besides `.look_msg`):

`@leave <exit>`	This is what the player sees when e leaves by an exit.
`@oleave <exit>`	This is what other people in the room see when a player leaves by an exit.
`@arrive <exit>`	This is what the player sees after passing through the exit and arriving at the new location.
`@oarrive <exit>`	This is what others at the destination see when the player arrives.
`@nogo <exit>`	This is what a player sees if he can't get through the exit for any reason.
`@onogo <exit>`	This is what other people in the room see if a player tries an exit but can't get through.

Pronoun and verb substitutions are addressed at length in the programming tutorial (see page 108), or you can read the online help text in help pronouns.

Briefly, %n substitutes the name of the player, %s is the subject pronoun (he/she/e/etc.), %p is the possessive pronoun (his/her/eir/etc.), %r is the reflexive pronoun (himself/herself/emself/etc.), %o is the object pronoun (him/her/em/etc.). These are the ones most commonly used for exit messages. There are others. Third person singular verbs, when surrounded by angle brackets (<>) and preceded by the percent sign (%) will agree with the gender of the player. Specifically, the messages will print correctly even if player's gender is set to plural. For example:

```
%N %<goes> through the door.
```

would yield:

```
Yib goes through the door.
```

but:

```
Bits go through the door.
```

Let's start with the east exit from the front room to the porch, then do the west exit from the porch to the front room. It's easiest to add the messages if you are in the room that is the exit's source, rather than the destination. (If you are elsewhere, you'll have to refer to the exits by their object numbers rather than by name.)

Starting in the Front Room:

```
@leave east is "You push open the screen door and head out
to the porch."
@oleave east is "%N %<pushes> open the screen door to the
east and %<heads> out to the porch.  The door slams shut
behind %o."
@arrive east is "The screen door slams shut behind you with
a bang."
@oarrive east is "%N %<comes> out through the screen door to
the west.  The door slams shut behind %o."
@nogo east is "You push on the screen door, but someone
seems to have nailed it shut."
@onogo east is "%N %<pushes> on the screen door, but someone
seems to have nailed it shut."
```

Now for messages on the exit going the other way. From the Porch:

```
@leave west is "You pull open the screen door and head into
the cottage."
@oleave west is "%N %<pulls> open the screen door to the
west and %<goes> inside.  You wince as the door slams shut
behind %o."
@arrive west is "The door slams shut behind you."
@oarrive west is "%N %<comes> in through the door to the
east.  You wince as it slams shut behind %o."
@nogo west is "You try the door, but it seems to be nailed
shut."
@nogo west is "%N %<tries> the screen door, but it seems to
be nailed shut."
```

A few comments:

It's good for the oleave (and sometimes oarrive) messages to mention the compass direction (if there is one). This helps others keep their bearings of where they are and what's beyond. It's especially good for when one player is following another.

Often it's sufficient to set only the leave message or only the arrive message, rather than both, such as when a player is going through an open doorway, for example. In the above example, I used the screen door slamming for the arrive message, to add to the effect.

If the exit is likely never to be locked or otherwise impassable, it's acceptable to omit the `nogo` and `onogo` messages.

Give some thought to who hears/sees what. I wanted to embellish the feel of the door slamming, and did that by having people wince, but didn't want to bombard them with it by using it in every single message. I chose to have people on either side of the door wince when someone goes in, but no one wincing when someone goes out. The choice was arbitrary, but deliberate. Paying attention to small details will give your work a richness that it might otherwise lack.

This may seem like a lot of work on top of `@dig` (voilà, you have an exit), but exit descriptions and messages give areas on the MOO a much stronger VR feel, and make any area more fun and interesting to explore. The messages don't have to be fancy, but they should be appropriate to the situation

Determining a Room or Object's Contents Definitively

Let's revisit the fact that every object has a property called `.contents`, which is either a list of object numbers or the empty list. Each of those listed objects will reciprocally have this object in its `.location` property.

Normally, when you look at a player, you will be presented with a list of things e is carrying, and when you look at a room, you will be told about things you see there. There are ways, however, for a programmer to camouflage or conceal what is in a room, container or player, and there are ways to circumvent such programming. First I will discuss camouflaging, then circumvention.

Integration – The basic room class shows you the room's description, then lists the non-player objects in it, then the players present:

```
The Conservatory
You are in a glass-sided room filled with orchids,
bromeliads, and other tropical plants.
You see trowel here.
Gardener is here.
```

A room class that supports object integration, on the other hand, will check for a special property on the objects within it and integrate that text into the description itself. The property usually has the name `.look_msg`. If our conservatory were an integrating room, and if the trowel had a `.look_msg` saying, “Off to one side is a trowel,” then the room’s description would look like this:

```
The Conservatory
You are in a glass-sided room filled with orchids,
bromeliads, and other tropical plants.  Off to one side is a
trowel.
Gardener is here.
```

Integrating objects into a room’s description has advantages and disadvantages. The main advantage is that the text is more pleasing to read, and many builders choose integration for this reason, rather than from any intent to deceive. The disadvantage is that the hapless explorer might now overlook the fact that the trowel is an object (which might be interactive or relevant to the scene at hand) *or* would have to spend time trying to examine the orchids and bromeliads which are merely a part of the description and not actual objects. A practiced MOOer *might* realize that in integrated rooms there is typically one sentence per integrated object, and therefore guess that “orchids, bromeliads, and other tropical plants” are so-called *tiny scenery* and that the trowel is a bona fide object, but there is no reliable way to tell for sure just by looking.

Darkness – Rooms have a `.dark` property, which, if set to a non-zero value suppresses the display of the room’s contents. Such a room might have a clue in its description about how to turn on the light, such as, `"As you grope around in the dark, your hand encounters a string,"` that, when pulled, turns on a light.[16]

The way to circumvent such techniques is to use the `@contents` verb. It isn’t as pretty, but it’s useful if you’re on the prowl for objects that might do something. Consider this example:

```
Undertaker's Cottage
This front room of the cottage is reminiscent of an old-
fashioned parlor, the kind one never actually went into.  At
one end, an overstuffed couch, at the other a stone
fireplace.  In between, six French Empire chairs, facing
each other gloomily, three and three.  Off in a corner sits
an ancient pump organ.  On one of the walls is a collection
```

[16] Here’s a bit of behind-the-scenes technology. It may not be of immediate interest if you are a beginner (and you can safely skip over it if it doesn’t make sense right now), but it may be of interest further down the road, or if you are an intermediate-level MOOer.

When you enter a room, that room’s `:look_self` verb is called. This verb in turn calls the room’s `:description` verb which assembles the text which will be displayed as the description, and it also calls the room’s `:tell_contents` verb which does the actual work of printing out what objects and players you “see” when you enter or look at the room. The `:tell_contents` verb calls the `:contents` verb which *usually* returns the value of the room’s `.contents` property BUT a room’s owner, can, if e wishes, cause the `:contents` verb or the `:tell_contents` verb to give incomplete or spurious information. In the case of a room with integrated objects, the `:description` verb tells you more (the integrated objects are integrated) and the `:tell_contents` verb tells you less (integrated objects are omitted, so as not to be mentioned redundantly). So in the quest for more pleasant prose, information is regrettably lost (i.e. which nouns in the description represent actual objects of possible interest).

```
of portraits.  Doors to the northeast and southwest stand
slightly ajar, as if someone were beyond them, watching...
or waiting.  Everything is covered with a layer of dust.
Cold stone steps lead down into darkness.
You see The Undertaker and Epitaph Registry here.
Yib is here.

@contents here
Undertaker's Cottage(#101792) contains:
The Undertaker(#666)   a fireplace(#78070)   pump
organ(#73881)   Collection of portraits(#14627)   Epitaph
Registry(#15588)   Yib(#58337)
```

If you were exploring this room to see what might be done here, you would examine the undertaker, the fireplace, the pump organ, the collection of portraits, and the epitaph registry. You wouldn't bother with the overstuffed couch, the empire chairs, the dust or the cold stone steps.

To summarize, then, objects can contain other objects. The contained objects are stored as a list of object numbers in a property named `.contents`. There is a `:contents` verb which usually returns a complete list of an object's contents, but which can also be programmed not to do so. The `@contents` command displays a definitive list of an object or room's contents, irrespective of other programming.