

yduJ's Programming Tutorial

This tutorial steps you through the creation of a MOO object and its verbs. There are three chapters: Chapter 1 programs the object in a straightforward manner. Chapter 2 adds bells and whistles to the program. Chapter 3 brings up

some issues that don't fit well into the tutorial framework. To read a chapter, type "read <chapter number> in tutorial" The chapters are organized into named sections; you may see the organization with "read index in tutorial", and each individual section may be read with "read <section name> in tutorial" You may read the entire thing with "read everything in tutorial" Comments about the tutorial should be directed to yduJ.

-- -- -- Introduction

This is a programming example of reasonable complexity. We are going to demonstrate property use, forking, and generic object use.

Our example is a wind-up toy. The basic things that a wind-up toy does is get wound, and then slowly wind its way down, hopping or rolling or twirling along.

First, let's create a couple of objects to work from. In this tutorial, material indented by two spaces was captured from a transcript, or is verbatim something you should type. In this case, the author typed the @create line, and the MOO responded with the text following. [Try 'help' on any command in this tutorial that you don't understand.]

```
@create $thing named Generic Wind-Up Toy,Toy
You now have Generic Wind-Up Toy (aka Toy) with object number #12221
and parent generic thing (#5).
```

```
@create #12221 named Wind-Up Duck,Duck
You now have Wind-Up Duck (aka Duck) with object number #12222 and
parent Generic Wind-Up Toy (#12221).
```

We'll refer to these as Toy and Duck, the aliases we gave in the @create command.

-- -- -- Simple Wind

First, we need some way to tell if the toy has been wound up. We'll create a property called "wound" on the generic toy.

```
@property toy.wound 0
Property added with value 0.
```

Before we can write our program, we need a verb... We want to give commands like "wind duck" so we give it an argument list of "this". The variable "this", when used inside a verb, refers to the actual object (e.g. the duck) on which the verb called. Its place in the argument list designates how the verb will be found by the built-in parser.

```
@verb toy:wind this
Verb added.
```

Now we can make a simple program to wind the toy.

```
@program toy:wind
this.wound = this.wound + 2;
player:tell("You wind up the ", this.name, ".");
player.location:announce(player.name, " winds up the ", this.name, ".");
.
```

Remember that "this" takes the place of whatever we will type in place of "this" in our command line, so when the program runs, it will change the .wound property of the duck. "Player" is the person who types the command. These are built in variables available in every program, they just take on different meanings depending on who types the command and what object is used in the command. The idea behind adding 2 to this.wound is that you can wind it a bunch of times to make it go longer.

Let's try it on our duck:

```
wind duck
You wind up the Wind-Up Duck.
```

Everyone else sees:

```
yduJ winds up the Wind-Up Duck.

-- -- -- Simple Messages
```

Next, we want to make the duck actually move around. Remember that we're doing all the programming on the generic toy, and using the specific instance of the duck to test things out. To be properly generic, we need to define different properties that can hold message strings for the motions of the toy. So let's define two messages. One for the toy to start moving, and another for it to print as it continues to wind down.

```
@property toy.startup_msg ""
Property added with value "".

@property toy.continue_msg ""
Property added with value "".
```

We named these with "_msg" in order for the messages to show up in @messages, and be settable with the @message_name object is "text" command. (See help @messages for more details.) Let's set values of these messages on the duck. While we're at it, we should give our duck a description!

```
@startup duck is "waddles about and starts rolling forward."
You set the "startup" message of Wind-Up Duck (#12222).
@continue duck is "swivels its neck and emits a >> Quack <<"
You set the "continue" message of Wind-Up Duck (#12222).
@describe duck as "A yellow plastic duck with wheels at the bottom and a knob
for winding."
Description set.
```

-- -- -- Simple Drop

We should be ready to roll now... We're going to have the wind up toy start up when the player drops it. This introduces another concept, that of specializing an existing verb. The generic thing (\$thing) defines the verb :drop. Normally, if we type "drop duck" the verb on \$thing gets invoked on the duck (even though it's a few levels removed from \$thing). If we define a :drop verb on the generic toy, we don't want to have to type in all the code from \$thing:drop; we just want to have our new code executed after the normal drop gets done. In order to get both things, we use the primitive pass(@args).

This is the slipperiest concept in MOO programming, yet the basis for the entire object oriented nature of the MOO. An easy way to think of pass is like another verb call: it just does the "basic thing" that this verb normally does. We can add our specialization code both before and after this basic thing. The @args part means just to use the same verb arguments in the base case as we do in our specialization. More sophisticated users of pass may wish to change these arguments; for now, just take it as gospel. Sometimes you won't want to use pass at all, if you don't want the basic thing to happen, but want to completely override the default behavior. This is OK, you just have to think about what you want when deciding whether and where to put the call to pass(@args).

We want the dropping to happen before any of our startup, so we call pass(@args) right away. We better check if the toy has been wound up, and not do anything special if it hasn't. Next, we print out our startup message, and schedule some tasks to be run a little later to print out the progress of our toy. Let's pick every 15 seconds to print out a message, and print out as many messages as we have in this.wound. Each time it actually prints a message, it will "unwind" a little, so we decrement this.wound. Note that the stuff between the fork and the endfork doesn't get done until the fork actually starts, and the forks start 15, 30, 45, etc. seconds after all the code in our drop verb has finished.

Our @verb command looks a little funny, because we're forced into using some more advanced features of MOO verbs. If you have a verb named foo*bar, then in order to invoke that verb, the user must type at least "foo", and may type any part of "bar" that they like; the * is putting in an abbreviation.

\$thing:drop has these abbreviations built in, so you can just type "d object" to drop something. Also, drop has a synonym, "throw" (more properly, th*row).

Putting both verbs in a double-quoted string is the way to say "two names for this verb". Really, any number of synonyms are possible. We'll continue to refer to this verb as "drop", even though we could just as truthfully refer to it as "throw". We use these names because we want to name our verb exactly the same as the one on \$thing, so people who prefer to use "throw" will still get our specializations.

```
@verb toy:"d*rop th*row" this
Verb added.
```

```

@program toy:drop
pass(@args);
if (this.wound)
  this.location:announce_all(this.name, " ", this.startup_msg);
  for x in [1..this.wound]
    fork (x * 15)
      this.location:announce_all(this.name," ", this.continue_msg);
      this.wound = this.wound - 1;
    endfork
  endfor
endif
.

```

It's worth noting that 0 in MOO is also "false" as well as being "zero", which is convenient. Thus we didn't have to use (this.wound!=0) in our if statement, but could use just (this.wound).

Now, let's drop our duck and see how it works!

```

drop duck
You drop Wind-Up Duck.
Wind-Up Duck waddles about and starts rolling forward.
Wind-Up Duck swivels its neck and emits a >> Quack <<
Wind-Up Duck swivels its neck and emits a >> Quack <<

```

It actually waited 15 seconds in between each of those messages. I tried it again, but this time I typed @forked right away to see that the tasks had been scheduled:

```

@forked
Queue ID      Start Time          Owner              Verb (Line) [This]
-----
1231283976    Jul 23 15:03:28 1991  yduJ              #12221:drop (6) [#12222]
577459244     Jul 23 15:03:43 1991  yduJ              #12221:drop (6) [#12222]

```

-- -- -- Introduction (2)

How could we improve on our wind-up toy? Or, rather, what are some of its problems?

What happens when someone picks up the toy while it's going? What if you try to drop it somewhere you're not allowed to? Perhaps we should only allow someone to wind it up if they are holding it. What if someone winds it five thousand times? Perhaps we should allow programmers to make more complicated messages on their child objects.

We'll address each of these issues in chapters 2.

-- -- -- Complex Wind

Let's start with requiring someone to be holding the duck in order to wind it. We take the code from before, and stick an if/else/endif around it.

```

@program toy:wind
if (this.location == player)
  this.wound = this.wound + 2;

```

```

    player:tell("You wind up the ", this.name, ".");
    player.location:announce(player.name, " winds up the ", this.name, ".");
else
    player:tell("You have to be holding the ", this.name, ".");
endif
.

```

That was simple enough... again 'this' is whatever wind-up toy was wound, and 'this.location' is where the toy is now. If it is in the player's inventory then this.location will be equal to the variable 'player'.

Let's complicate the code once more, and then we'll be done with the :wind verb. We should have a maximum number of turns the toy will allow, otherwise a malicious player could spend a few hours winding the toy, and then unleash it on an unsuspecting public. If we wanted to be really fancy, we could make the toy break if they wound it too far, and require then to repair it with a screwdriver (which we would create for the purpose), but let's leave that as an exercise for the reader.

We create a property for the maximum number of turns, so different toys can have different maximums. We'll give it a default value of 20, though, so it doesn't have to be set for each new toy.

```

@property toy.maximum 20
Property added with value 20.

```

Now we insert another set of if/else/endifs inside our current set.

```

@program toy:wind
if (this.location == player)
    if (this.wound < this.maximum)
        this.wound = this.wound + 2;
        player:tell("You wind up the ", this.name, ".");
        player.location:announce(player.name, " winds up the ", this.name, ".");
        if (this.wound >= this.maximum)
            player:tell("The knob comes to a stop while winding.");
        endif
    else
        player:tell("The ", this.name, " is already fully wound.");
    endif
else
    player:tell("You have to be holding the ", this.name, ".");
endif
.

```

In order to add to the feel of the toy, we put in another test: if, after adding 2 to the wound property, it has reached the maximum, we tell the player they've come to the end. The more completely you can describe an object's actions and responses to actions, the richer the feel of the Virtual Reality.

-- -- -- Complex Messages

Earlier we showed how to make user settable messages on an object. Now we will show how to enable another programmer make more complicated messages on our windup toy.

A simple yet very useful method is to define a verb for each message property, which just returns that property, and then have all the uses of that property use the verb instead. For example:

```
@verb toy:going_msg this none this
@program toy:going_msg
return this.going_msg;
.
```

However, it seems almost silly to make a whole bunch of verbs, one for each property... And indeed it is. The built-in variable "verb" is always set to the name that this verb was invoked with, so we can get all the messages in one verb, with this idiom:

```
@verb toy:"going_msg wind_down_msg continue_msg startup_msg" this none this
@program toy:going_msg
return this.(verb);
.
```

There's a lot of stuff to explain in this little bit of code! First, the syntax of the @verb command, with the several messages listed in a quoted string, indicates that each of those names is an alias for the verbnam, and the verb can be invoked by any of those names. Second, the builtin variable "verb" is set to whichever name was actually used. This enables the property reference to be a simple construction from the verb name, using the expression syntax for property references, <object>.<expression>, where in this case expression is a simple variable reference.

Here's an example of how we can customize just one of those messages programmatically with the wind-up duck. In the continue_msg we'll make the duck quack once, twice, or thrice, randomly. We leave the other messages alone, as we'll just be using static text.

```
@verb duck:continue_msg this none this
@program duck:continue_msg
times = {"once","twice","thrice"}[random(3)];
return "swivels its neck and quacks " + times + ".";
.
```

-- -- -- Complex Drop

Let's tackle the drop verb now. There are a lot of problems. First, what happens if someone picks up the toy while it's still going? The messages still appear! What should happen? Two ideas come to mind: 1. It should fully discharge its windings, as real life windup toys are wont to do; 2. It should just stop running, as though the player had grabbed the windup knob to prevent it from further discharge. We'll choose option 2 for our example. Option 1 would be easiest to do by specializing the existing "take" verb to reduce the .wound property to 0, printing appropriate messages.

The major problem with our drop verb is that it schedules all its tasks at one time. Once scheduled, they *will* run, whether or not the situation has changed. To solve this problem, we only schedule one task at a time, making it the job of that task to figure out whether it has wound down, and schedule

another task. It should also check if it has been picked up; if so it should neither print messages nor schedule the next task.

Having called this section "Complex Drop", we're going to defer the complexity to an auxiliary verb, and make Drop itself simpler.

```
@program toy:drop
pass(@args);
if (this.wound)
  this.location:announce_all(this.name, " ", this:startup_msg());
  fork (15)
    this:do_the_work();
  endfork
endif
.
```

We will use the special arguments "this none this" in our @verb command. This is a special kluge that means this verb is not a command, and cannot be typed at the command line. It also won't show up in @examine commands. This helps to keep down the clutter of @examine, and because this verb is only an internal function, it wouldn't make sense to type as a command line.

```
@verb toy:do_the_work this none this
@program toy:do_the_work
if (this.wound)
  if ($object_utils:isa(this.location,$room))
    this.location:announce_all(this.name," ", this:continue_msg());
    this.wound = this.wound - 1;
    if (this.wound)
      fork (15)
        this:do_the_work();
      endfork
    else
      this.location:announce_all(this.name, " ", this:wind_down_msg());
    endif
  endif
  if (this.wound < 0)
    this.wound = 0;
  endif
endif
.
```

Let's go through this step by step. First we check if we're still wound up. Theoretically this should always be true, but it's best to check. Assuming .wound is true (that is, nonzero), we check to see if we're in a room. The utility verb \$object_utils:isa(x,y) checks to see if an object X will behave as

an object Y, that is, if Y is in the object's ancestor tree. We do this because we only want to make our noises if we're wandering around in a room.

A

player won't be a room, and so won't pass this test. Now, we print our message, and decrement this.wound to say we've wound down once. Next we think about scheduling the next task. We check this.wound again, because if we subtracted the last bit, we don't want to bother scheduling it. Note that we call this very same verb, this:do_the_work. Just for extra feeling, we put in a new message to be printed when the toy has actually wound down. Oops! Better add that property... And finally, at the very end, we've gotten all

anal retentive with error checking and made sure that this.wound never gets into negative numbers. (Notice that if it *did* get into negative numbers, it would never stop. So this check is not entirely pointless!)

```
@property toy.wind_down_msg ""
Property added with value "".
```

```
@wind_down duck is "hiccups once and stops rolling."
You set the "wind_down" message of Wind-Up Duck (#12222).
```

-- -- -- Descriptions

It might be nice if you could tell by looking at the toy that it was moving, and not have to wait for it to tell you so. Like "drop", this is done by specializing an already existing verb, called description, by calling pass(@args) and then doing some additional stuff. There are a number of relevant differences between this and most other verbs, however.

Most verbs do something, that is, when you type a command, it produces a set of messages, and changes some properties on some objects. Some verbs, however, are not commands, and don't even print any messages. Instead, they return a value to the calling verb, for that verb to use to produce messages or change properties. These are analogous to "functions" in other languages. Description is one such verb. By default, that is, in the basic case, it returns the property 'this.description'. Verbs like 'look' call description on objects and print out the strings they return. So when we specialize description, we won't be calling pass(@args) all by itself, and then printing out more text, instead we'll call pass(@args), add text to that, and return the whole pile.

Again because we're trying to write generically, we create a property to hold a message to be added to the description when the toy is running, and we set it for the duck.

```
@property toy.going_msg ""
Property added with value "".
```

```
@going duck is "The duck is rolling forward with a slight waddle."
You set the "going" message of Wind-Up Duck (#12222).
```

The description verb again needs the "this none this" args, since it is just an internal verb called by programs.

```
@verb toy:description this none this
Verb added.
```

```
@program toy:description
basic = pass(@args);
if (this.wound && $object_utils:isa(this.location, $room))
  return basic + " " + this:going_msg();
else
  return basic;
```

```
endif
```

```
.
```

First we call `pass` to get the underlying description. We store away that text in a variable `'basic'`, because we'll need it later. Then we check to see if we're wound up, and also to see if we are in a room. If we aren't in a room, even if we're wound, we shouldn't say we're moving. If we are, though, we return the basic information plus our `going_msg`. We put in a couple of spaces to separate the sentences. Otherwise, we're quiet, and we just return the basic string. What does it look like?

```
look duck
```

```
A yellow plastic duck with wheels at the bottom and a knob for winding. The duck is rolling forward with a slight waddle.
```

Note that here and in a few other places I've used things like `$object_utils`. These things starting with `$` are utility objects, part of the "core database". There are a lot of quite useful functions on these core objects.

```
-- -- -- Permissions
```

To allow other people to create their own wind-up toys, we set the generic toy world readable:

```
@chmod toy r
Object permissions set to r.
```

This isn't really good enough, due to the screwinesses of the MOO permission system. Generally speaking, when you create an object, you become the owner of all its properties as well. This is called "having c permissions". But that means that someone else (including the owner of the `*verb*` on the object) can't change the property. To get around this problem, we set the wound property `!c` (meaning "not c"). Then the wound property always belongs to the owner of the generic toy, and not to the creator of any specific instance.

```
@chmod toy.wound !c
Property permissions set to r.
```

It tells us what the permissions ended up being, after stripping off the "c", which is "r" for readable.