LambdaCore Database Programmer's Manual

For LambdaMOO version 1.3
July 1991

Mike Prudence (blip)
Simon Hunt (Ezeke)
Floyd Moore (Phantom)
Kelly Larson (Zaphod)
Al Harringtom (Geezer)

Introduction

The LambdaCore database provides the facilities needed to make a LambdaMOO server useful for Multi User Adventuring. If you compare the LambdaMOO server to a piece of computer hardware, then LambdaCore is the operating system needed to allow the user to do useful work.

This document gives a rundown on the elements of the LambdaCore database, and provides details of each of the verbs used by the database. It also provides source for some of the verbs, to aid understanding and provide extra information.

The user is assumed to have read the two companion manuals to this one, *The LambdaMOO Programmer's Manual* and *The LambdaCore User's Manual*. An understanding of MOO concepts, the MOO language and the facilities provided by the LambdaCore database is needed to follow the explanations given in this document.

1 The LambdaCore Classes

The LambdaCore database provides several basic classes that are used to define the virtual world. It is these classes that provide the *core* of any MOO database built on the LambdaCore database. As well as defining obvious things like players, rooms and exits, the classes also provide all the commands used to interact with the database, and several utility functions for use by the MOO programmer writing his or her own verbs.

The LambdaCore classes can be divided into two categories:

- Generics: These are classes containing generic objects, for example players, rooms, exits and so on. If a new generic object is created, it would be placed in this list.
- Utilities: These are classes containing utility verbs that can be used by other verbs. Example include string utilities, trig functions etc.

The sections that follow describe the LambdaCore classes in detail, providing insight into what each verb and property is used for, and some idea of how they should be used in your own MOO programs.

1.1 The LambdaCore Generic Classes

The following generic classes are available in the LambdaCore database:

`The Root Class ($root_class)`
: This is the basic class from which everything else in the database is descended. It defines basic operations and properties on an object.

`The Generic Room ($room)`
: This class is the parent for all the rooms in the database.

`The Mail Distribution Center ($mail_room)`
: This is a special class used to define a room for dealing with MOO mail. It is a child of the generic `Room` class.

`The Generic Mail Recipient`
: To be written.

`The Generic Editor's Office ($editor)`
: This is a special class used to define a room used for program editing. It is a child of the generic `Room` class.

`System Object (#0)`
: This is the keystone object for the database. It holds various important system properties, as well as pointers to the various other classes and objects that are system-wide.

`The Generic Thing ($thing)`
: This class defines a TinyMUD style object, that is, something that can be picked up and put down. It defines various messages related to taking and dropping the object.

`The Generic Container ($container)`
: This is a child class of the `Thing` class. A container is an object that can have other objects placed within it.

`The Generic Note ($note)`
: A note is a child class of the `Thing` class. It is used to store text messages, which may be encrypted. A note can only be recycled by the owner.

`The Generic Letter ($letter)`
: A letter is a child class of the `Note` class. It provides the same facilities, but also has a `burn` verb which the recipient can use to recycle the letter.

`The Generic Player ($player)`
: This is the class from which all players are descended.

`The Generic Programmer ($prog)`
: This is a child of the `Player` class used for players that are MOO programmers. It defines a set of verbs that are useful for programming activities.

`The Generic Wizard ($wiz)`
: This is a child of the `generic programmer` class, used for wizards.

`The Generic Exit. ($exit)`
: This class defines a generic exit used to link one object of class `Room` to another object of class `Room`. It defines a number of messages that may be generated when an exit is used.

1.1.1 The Root Class

The root class is the base class from which all objects are descended. It is the keystone of the database and defines fundamental verbs and properties common to every object. If you are a programmer (ie, you have a player object with the *programmer* bit set) you can examine the properties and verbs of the root class using the command

```
@show $root_class
```

You can examine the code for a verb on the class by using, for example, the following command.

```
@list $root_class:description
```

This lists the program definition for the verb `$root_class:description`. An interesting point to note is that this code can be changed by the owner - in this case the Wizard - to provide any functionality desired. This configurability of the basis of the whole LambdaCore database allows a large degree of flexibility in the way the LambdaMOO server is used. It also allows for very subtle and perplexing problems. Care must be taken when editing definitions on any of the fundamental classes, the `$root_class` in particular. However, as the base classes of the LambdaCore database have been thoroughly tested and debugged, there should be very little need for any changes by the average database administrator.

The following section lists the verb and property definitions for the `$root_class`. For each verb, a description of it's function is given, along with any interesting points. The intent is to present each verb in such a way that it is possible to grasp the whole picture, rather than just looking at the individual brushstrokes.

`string` **description** () Verb

The `:description` verb on any object is supposed to return a string or list of strings describing the object in the detail someone would notice if they were specifically looking at it.

The default implementation of the `"look"` command (defined on the `$room` class), prints this description using the `:look_self` verb on the object. `:look_self` uses `:description` to obtain the text to display.

`none` **describe** (*value*) Verb

The `:describe` verb is used to set the description property of an object. This is only allowed if we have permission, determined using the `$perm_utils:controls()` verb. By overriding this verb and the `:description` verb, it is possible to completely change the representation of an object description. This is done invisibly to anyone outside the object, as long as you adhere to the same interface to `:description` and `:describe`.

`none` **look_self** () Verb

The `:look_self` verb on any object is used to `:tell` another object what this object looks like, in detail. This verb makes use of the `:description` verb on the object to obtain a string or list of strings to print. It would be possible to override this verb to produce a special description for an object. However, any other verbs that use the `:description` verb of the object will not see the extra information added by the overriding function. The `$room` class overrides this verb with code to print the room `name` and a list of objects that are in the room.

`none` **tell** (`string` *strings*, . . .) Verb

This verb is used to send a message from one object to another. The `$root_class` definition of this verb tests to see if the object is a player, and if it is, uses the `notify` primitive to print the argument list on the player's screen, if they are connected. However, this verb can be overridden to allow arbitrary objects to pass messages between each other, or to augment the way the message is handled.

One simple example is that of an object that listens to everything that happens in a room. Every verb that needs to send text to players uses the `:tell` verb. If an object has it's own `:tell` verb, it too will be able to act upon the messages sent between objects in a room.

The `$player` class overrides this verb to filter messages in two different ways, as show below:

```
if (typeof(this.gaglist) != LIST || !(player in this.gaglist))
  if (player != this && this.paranoid == 1)
    pass("<", player.name, "(", player, ")> ", @args);
  else
    pass(@args);
  endif
endif
```

Firstly, if the message comes from a player that we don't want to listen to - the player has been *gagged* - then the message is thrown away. Secondly, if the player is being *paranoid*, and the message is not from ourselves, it is prefaced with the name of the originating object. The `pass` primitive is used to allow the `:tell` verb of the parent class to send the message after it has been modified.

`none` **moveto** (`obj` *where*) Verb

This verb is used to change the location of an object to be *where*. The built-in function `move()` is used by this verb. Any error codes generated by it are returned by this function. This verb is intended to be used by any other verbs that must move an object to another location.

One important point to note is that this uses the `set_task_perms()` primitive to set the task permissions to those of the thing that is being moved.

Again, by overriding the verb definition on an object, it is possible to augment or change the way an object is moved. For example, you could keep a list of places visited by simply recording the *where* objects in a list every time this function is called.

`num` **accept** (`obj` *thing*) Verb

This verb is used to control what objects are permitted to be placed *inside* other objects. If this verb returns '`0`', then the object *where* cannot be moved into this object. Conversely, if the verb returns a non-zero value, the object is allowed to be placed inside this object. The verb is called by the server when it executes a `move()` primitive.

The `$root_class` definition returns a zero value. In this case, no objects are allowed inside any objects that are children of the `$root_class`. The `$room` class definition provides for a flexible scheme using various different criteria, as shown in the following code :

```
what = args[1];
return this:is_unlocked_for(what) &&
       (this.free_entry ||
       what.owner == this.owner ||
       (typeof(this.residents) == LIST && what in this.residents));
```

Starting at the top of the conditional expression, we see the locking condition being checked. If the room lock forbids this object to enter the room, then the `:accept` verb returns zero.

If this is not the case, then we consider the value of the `free_entry` property. If this is set to a non-zero value, then the object is allowed to enter the room.

If the owner of an object is the owner of a room, the object is allowed to enter.

Finally, if a `residents` list is defined in the room, and the object is in the list, then it is allowed to enter.

This complex set of conditions shows how an arbitrary set of criteria can be applied to the movement of objects into other objects.

`obj` **match** (`string` *name*) Verb

This verb is used to find things that are located within this object. It tries to match *name* to something in the contents list of this object, using object names and object aliases. This verb uses the `$string_utils:match()` verb to do the actual searching. If a match is found, the object that matched is returned. If more than one object matches, then `$ambiguous_match` is returned. If no match is found, then `$failed_match` is returned.

`none` **exam*ine** () Verb

This prints out some useful information about the object to the player. It is provided as a player command, to allow every player to determine basic information about any other objects they come across. For example,

```
>exam #0
The System Object (#0) is owned by Wizard (#2).
Aliases:  The, Known, and Universe
(No description set.)
```

The idea is to allow every player to discover the owner, full name, description and aliases of any object.

If you control the object, the lock for the object is shown. If the object has other objects inside it, then the contents list is printed out, too. If the object has verbs defined on it, then these verbs are listed, provided they are readable, and have not been hidden by setting the argument specifiers to the triplet '`this`', '`none`' '`this`'.

`none` **tell_lines** (`list` *strings*) Verb

This outputs out the list of strings *strings* to the object, using the `tell` verb for this object. Each string in *strings* is output on a separate line.

`num` **set_name** (`string` *value*) Verb

This verb sets the name of the object to *value*. It returns '1' if the name was set to the *value* successfully, otherwise it returns '0'. This verb, and the `:title` verb are used to control access to the `name` property of an object.

`num` **is_unlocked_for** (`obj` *thing*) Verb

Returns '1' if the object is unlocked for the argument. If the value of `this.key` is zero, the object is unlocked. If this is not the case. the verb `$lock_utils:eval_key()` is used to determine the result.

`string` **title** (). Verb

This verb is used to get the `name` property of this object.

One example where it might be useful to redefine this verb is if you want to add an honorific or descriptive phrase to the end of your name. By overriding the `:title` verb, you can append anything you like to the `:name` property of the object.

`none` **recycle** () Verb

This verb contains no code for the `$root` class. It is called by the `recycle()` primitive just before an object is recycled. This is useful to make sure that recycling objects does not leave the database in a strange state. For example, the `$exit` class uses the `:recycle` verb to remove the exit from the the *entrance* and *exit* lists of its destination and source rooms.

`string` **titlec** () Verb

This verb performs the same function as the `title` verb, but returns a capitalised version of the `name` property of the object, using the `$string_utils:cap_property` verb.

`none` **eject** () Verb

This verb is used to remove something from the contents of an object. The owner of an object, or a wizard, can use this verb to eject a victim from inside the object. The victim is sent to `$nothing`, for most objects, or to `$player_start` if the victim is a player.

The root class defines a few basic properties that every object has. The verbs discussed above make reference to these properties - as well as the built in properties `name`, `owner`, `location`, `contents`, `programmer`, `wizard`, `r` and `w`. The following properties are defined for the root class:

key Property

This is used to determine what, if anything, an item is locked to.

aliases Property

This gives a list of the alternative names that can be used to refer to an object. The value of this property should be a list of strings, the various alternatives for naming the object. Note that the name of the object in question should be included in this list, because the built in name matcher uses only the `aliases` property when searching for matches.

description Property

This is a string or list of strings giving the description of the object - what we see when we `look` at the object.

1.1.2 The Room Class

The `Room Class` is the basic class from which all the *rooms* in the virtual world are descended. It is one of the very basic classes essential to constructing a virtual world; an `exit` is the other essential class.

A room can be thought of as a container for players and objects. It can have a number of exits that connect it to other rooms. These exits are directed; they lead *from* one room *to* another room. For a two way passage to exist, two exits are needed, going in opposite directions.

The room class defines a lot of verbs, which are used to provide an interface to the properties of the room.

One special point is worth noting about rooms and exits. An exit can have an arbitrary name - indeed, this is the usual case. In order for the room to recognise the exit name, and match it up with an exit that exists in the room, some form of *catchall* mechanism is used. If a player types a sentence that the parser cannot match with anything, it executes a verb called `:huh` in the current room, if one exists.

When this happens, the `:huh` verb is free to take the player's sentence, search for a valid exit, and act accordingly. This mechanism provides a very flexible arrangement for dealing with exits, and also allows a degree of player help to be added. If a close match to a command is found, the `huh` verb could detect this and provide a useful response that would help the player construct the correct sentence.

The following section lists the verbs defined for the `$room` class. Those verbs which are more properly described as player commands are detailed in a preceding section.

`none` **confunc** () Verb

This verb is called by the LambdaMOO server when a player connects in a room. The action coded into the `$room` class simply shows the player what the room looks like (using the `:look_self` verb of the room) and tells everyone else in the room that the player has connected.

`none` **disfunc** () Verb

This verb is called by the LambdaMOO server when a player disconnects from the game. The `$room` class definition of this verb moves the player home if s/he is not already there, using the `:moveto` verb on the player. One possible enhancement of this verb, already implemented in some MOOs, is to include a time delay between disconnection and movement of the player to his/her home. This would allow some tolerance of disconnection due to network problems.

`none` **say** (`string` *message*) Verb

This verb provides one of the basic ways in which players communicate. The action of the `:say` verb is very simple: it `:tell`s the player what s/he has just said, and tells everyone else what the player said. The text spoken is passed to all the objects in a room, not just the players, through the `:tell` verbs on the objects in the room.

By overriding this verb, it is possible to provide all sorts of effects that work on everything said in the room. For example, you could redirect messages to other rooms, or repeat messages to provide cavernous echoes.

`none` **emote** (`string` *message*) Verb

This verb is used for the *pose* type of interaction with other players. It functions in a similar way to the `:say` verb, but instead places the player's name at the front of the text. The actual output is done in two stages. The `:emote` verb is responsible for telling the player the action s/he has just performed. The `emote1` verb is then called to tell the other objects in the room of the `pose` action. This provides a two stage mechanism; either or both of the verbs can be overridden to provide special effects.

`none` **emote1** (`string` *message*) Verb

This verb is invoked by the `emote` verb to send the *emote* text to the other objectss in the room. By overriding this verb, it is possible to create special effects for `emote` actions that are only seen by the other people in a room. `:emote1` uses `$room:announce` to send it's message.

`none` **announce** (`string` *message*) Verb

This verb is a general purpose verb used to send a message to every object in the room except the player that invoked it. This is intended to be the way other verbs pass messages to objects in a room. For example, when an exit is activated, it uses `:announce` to inform the other players in the room of what has happened.

`none` **announce_all_but** (`list @`*args*) Verb

This general purpose verb is used to send a message to everyone in the room *except* a particular object. This can be used in situations where we wish to present one view to the world in general, and another view to a particular object, normally a player. Another common usage is to prevent robots that trigger actions based on a redefined `:tell()` verb on themselves from recursing, using something like

```
place:announce_all_but(this, "message");
```

`none` **huh** (`string` *sentence*) Verb

This verb is a stub used to call the `huh2` verb. It is called by the LambdaMOO server when it can't match a sentence given to it by a player. The server calls `:huh` with `verb` equal to the actual verb in the erroneous command line. This means it is not possible to use `pass()` if you override `:huh` in a room; it would pass control up to a verb on the parent named `verb`, i.e., whatever the verb was on the command line. This, by definition, doesn't exist. To get around this, and allow you to use `pass()` to get the default `:huh` behaviour, `:huh` calls `:huh2`. You should override `:huh2` if you wish to be able to use `pass()` to get the default `:huh` behaviour.

`obj` **match_exit** (`string` *exit*) Verb

This verb is used to determine if *exit* is the name of an exit leading out of the room. It performs a simple string match on the names and aliases of the objects in the `exits` list stored as a property of the room. The intent here is to allow for more sophisticated matching algorithms to be implemented. One might even go so far as implementing a fuzzy matching scheme, to allow for player misspellings. If a successful match is made, this signifies that an exit with the name *exit* leads from this room. It's object number is returned. If more than one match is found the value `$ambiguous_match` is returned. If no match is found, the value `$failed_match` is returned.

`num` **add_exit** (`obj` *exit*) Verb

This verb is used to add *exit* to the list of exits leading out of the room. This verb, and the `:match_exit` verb provide the interface to the room exits list. The way in which exits are stored, removed and matched has been separated from the interface so that different implementations of the *exits* concept can be used in different sub classes of the `$room` class.

If it is not possible to add *exit* to the room's exit list (normally because the object that invoked the verb does not have the required permission) then the verb returns '0'. Otherwise, a successful addition returns '1'.

`none` **tell_contents** () Verb

This verb tells us what things are visible in the room. It goes through the `contents` list of the room, and if it is not dark, prints the name of the object in a nicely formatted way. Three different formats are available depending on the value of the `ctype` property of the room. These are best illustrated by example.

Consider the a room in the LambdaCore database. With a `ctype` value of three (the default), the `tell_contents` verb produces the following output:

```
You see a newspaper and a fruitbat zapper here.
Wizard is here.
```

This format provides the separation of player objects from other objects, and provides the list in a way that fits in with the idea of a virtual reality. It is easy to read, and looks natural.

If the `ctype` value is changed to 2, the following is printed:

```
You see Wizard, a newspaper, and a fruitbat zapper here.
```

This format treats players and objects the same, and is useful if you wish to hide the fact that an object is not a player, or vice versa.

With a `ctype` value of one, the following is seen:

```
Wizard is here.
You see a newspaper here.
You see a fruitbat zapper here.
```

This format provides the advantage of having each item on a separate line, although it does mean that rooms with a large number of objects in might have excessively long contents lists.

Finally, with a `ctype` of zero, the following is seen:

```
Contents:
  Wizard
  a newspaper
  a fruitbat zapper
```

This is the sort of listing obtained in traditional TinyMU* games. It benefits from clarity, but is not as natural as any of the other forms.

If a value of `ctype` is set that is outside of the range zero thru three, then no contents list is printed. This can be useful if you want to handle the room contents listing as part of the description of the room. Also, if the `dark` property of a room is set to a non-zero value, then no contents list is printed.

As usual, this verb can be overridden to provide special effects. For example, you could apply a filter so that certain objects do not appear in the printed contents of a room, even if they appear in the contents list. This can be use to hide objects, for example, as part of a puzzle, or to vary how objects are seen, for example if you are looking through water at something.

`obj` **match_object** (`string` *name*) Verb

This is the verb used to search the player's locale for an object that has the name or pseudonym *name*. This verb handles mapping of `me` to the player object, `here` to the player's location as well as the use of `#<number>` to refer to a particular object. If none of these cases match, the verb searches the room contents and the players contents (or possessions) for a match. If a match is found, then the unique object number is returned. If *name* matches more than one object, then `$ambiguous_match` is returned. If no match is found, then `$failed_match` is returned.

The verb `:match_object` is the one to use to map names of objects to object numbers, when referring to objects that the player is able to *see* in his current location. This includes objects that the player might be carrying, but does not include objects that are *contained* in other objects.

`none` **@exits** () Verb

This verb is a player command used to print a list of the exits in a room. It can only be used by the owner of the room. The verb simply runs through the list of defined exits, stored in the property `exits`, and prints the exit name, object reference number, destination name, and exit aliases.

`none` **look_self** () Verb

This verb overrides the `$root_class` definition of the verb `:look_self` in order to provide a fuller description of a room than the `description` property gives. This verb prints the room name, followed by the room's `description` property, and then the list of contents of the room, using the room's `:tell_contents` verb. This is what the player would see if he was looking at the room.

The `description` property of the room is actually printed by using the `pass()` primitive to invoke the parent verb `:look_self`). Changes in the way an object's description is stored by the root class are invisible to this verb, because of the way `pass` is used.

`num` **accept** (`object` *thing*) Verb

This verb overrides the `$root_class` definition. The `$room_class` definition provides for a flexible scheme using various different criteria. The idea is to allow the builder flexibility in preventing objects from moving into the room. The following code shows how the `$room:accept` verb decides whether to allow *thing* to enter the room or not.

```
what = args[1];
return this:is_unlocked_for(what) &&
       (this.free_entry ||
       (what==this.blessed_object&&task_id()==this.blessed_task) ||
       what.owner == this.owner ||
       (typeof(this.residents) == LIST && what in this.residents));
```

Starting at the top of the conditional expression, we see the locking condition being checked. If the room lock forbids this object to enter the room, then the `:accept` verb returns zero.

If this is not the case, then we consider the value of the `free_entry` property. If this is set to a non-zero value, then the object is allowed to enter the room. This is used to provide public places, where anything and everything is allowed to enter. The default value of this property is `1`.

If the object has been *blessed* for entry into this room, and the task that is moving the object is the same as the task that requested the blessing, then the object is allowed to enter. Refer to the `:bless_for_entry` verb for further details on the concept of blessed objects.

If the owner of an object is the owner of a room, the object is allowed to enter. This is a general rule used as a catch-all case. It can be overridden by specifically locking against an object, but as a rule, it is desirable for objects owned by the room owner to be allowed to enter.

Finally, if a `residents` list is defined in the room, and the object is in the list, then it is allowed to enter. This concept can be used to define a set of objects that are allowed to enter the room without specifying a long and complex locking condition.

num **add_entrance** (`obj` *entrance*) Verb

This verb functions similarly to the `:add_exit` verb, but applies to `$exit` objects that lead *into* the room. If we imagine an `$exit` object as a flexible tube connecting two rooms, then the concept of specifying both ends of the tube seems natural. It is not usual to search the entrance list for a match, as you would with the exit list, but the concept of an entrance is included to cover unexpected cases.

If it is not possible to add *entrance* to the room's entrance list (normally because the object that invoked the verb does not have the required permission) then the verb returns '`0`'. Otherwise, a successful addition returns '`1`'.

`none` **bless_for_entry** (`obj` *thing*) Verb

This verb is called by an exit to allow an object special permission to enter a room. Two properties on the room store *thing* and the `task_id` of the calling task. The idea behind *blessed* objects is to allow an object temporary permission to enter. The permission is only granted if the request is made by an object that is an entrance into the room.

The idea here is that a normally inaccessible room can be entered providing you use an exit that leads into the room. In addition, the task ID of the task that asked for the *blessing* is stored, so that there is no way an object can become blessed, and then later gain entry to the room. The object being blessed is only allowed to enter once per blessing. Once the object has moved into the room, it's *blessed* status is removed by resetting the `blessed_object` property in the room to `$nothing`.

`none` **@entrances** Verb

This verb is a player command used to list the entrances to the player's current location. Only the owner of a room may list it's entrances. For every object kept in the room's `entrances` list, the exit name, object reference number and aliases are displayed to the player.

`none` **go** (`list` *args*) Verb

The verb is a player command used to move from one room to another. One special feature of this command is that it allows the player to string together a number of different movement commands into one use of the '`go`' command. This can save typing when we know where we want to go. The code for this verb is shown below:

```
for dir in (args)
  exit = player.location:match_exit(dir);
  if (exit < #0)
    player:tell("Go where?");
  else
    exit:invoke();
  endif
endfor
```

This simply moves through the list of exits given in *args*, matching the exit name in the current room, and invoking the exit if a valid match is found.

`none` **look** () Verb

This verb is a player command used to look at various things in the player's current location. It can be used meaningfully in other verbs, however, if the verb is called with no arguments. In this case, it calls the `:look_self` on the room which the verb was invoked on. You could use this verb to print the description of a room you are not in, but it is stylistically better to use the `:look_self()` verb of the room.

If a preposition is supplied, via this verb being invoked as a command, and the preposition is not '`on`' or '`in`', then the verb attempts to match the direct object with something in the room. If the match succeeds, the `:look_self` verb of the matched object is called to tell the player what the object looks like.

If the preposition is '`on`' or '`in`', then the player wishes to look inside a container of some sort, be it a member of the `$container` class, or in a player's inventory, for example. An attempt is made to match the indirect object with something in the room. If it succeeds, then an attempt is made to match the direct object with something inside the container previously matched. If this final match is made, the `:look_self` verb of the matched object is invoked to print it's description.

If the direct object is the empty string, '`""`', then the container's `:look_self` verb is called to print it's description.

Any ambiguous or failed matches produce suitable error messages.

`none` **announce_all** (`list` @*args*) Verb

This verb is another general purpose verb used to send a message to every object in the room. It is used for messages that we wish everyone to see, with no exceptions.

`none` **enterfunc** (`obj` *thing*) Verb

This verb is invoked by the LambdaMOO server when an object moves into a room, as part of action of the `move` primitive. The actions taken for a room are very straightforward. If *thing* is a player object, then we tell the player where s/he has moved into using the `$room:look_self` verb on the room. If the object is the *blessed_object* for this room, then the `blessed_object` property for the room is reset to `$nothing`. For further details on *blessed objects*, refer to the description of `$room:bless_for_entry`.

`none` **exitfunc** (`obj` *thing*) Verb

This verb is invoked by the LambdaMOO server when an object is moved out of a room, as part of the action of the `move` primitive. The action defined for the `$room` class is to do nothing.

This verb, and the `:enterfunc` verb, can be used for a variety of effects that need to take note of objects moving in and out of rooms. For example, consider a *torch* object that casts light on it's surroundings. When the torch is moved out of a room, the light that is casts moves with it. This can be tackled using the room's `exitfunc`, which could check if the object leaving is a torch, and if it is, the room could become dark. Similarly, when the torch enters a room, the `enterfunc` can be used to detect this, and brighten the room accordingly.

`num` **remove_exit** (`obj` *exit*) Verb

This verb performs the opposite function to the `:add_exit` verb. It removes *exit* from the room's list of exits. If it is not possible to remove *exit* from the room's exit list (normally because the object that invoked the verb does not have the required permission) then the verb returns '`0`'. Otherwise, a successful addition returns '`1`'.

`num` **remove_entrance** (`obj` *entrance*) Verb

This verb performs the opposite function to the `:add_entrance` verb. It removes *entrance* from the room's list of entrances. If it is not possible to remove *entrance* from the room's entrance list (normally because the object that invoked the verb does not have the required permission) then the verb returns '`0`'. Otherwise, a successful addition returns '`1`'.

`none` **@add-exit** Verb

This is a player command used to add an exit to the current room. This is normally used when someone else has created an exit they want to lead out of a room you own. The verb matches the direct object string with an object in the room to get the object reference number for the exit. If the object found is not a descendant of the `$exit` object, the verb is aborted with an error.

Otherwise, if the destination of the exit is readable and leads to a valid room, an attempt is made to add the exit using the room's `:add_exit` verb. If this fails, a suitable error message is sent to the user.

`none` **@add-entrance** Verb

This is a player command used to add an entrance to the current room. This follows much the same sequence as for the '`@add-exit`'. An attempt is made to match the direct object supplied with an object in the room. If this fails, the verb is aborted with a suitable error message.

Otherwise, if the object found is a descendant of the `$exit` class, then the exit is checked to make sure it goes to this room. If this is the case, then the exit is added as an entrance using the room's `:add-entrance` verb.

`none` **recycle** () Verb

This verb is called by the LambdaMOO server when a room is recycled. When a room is recycled, something has to be done with the room's contents, both players and objects, to stop them ending up in `$nothing`. This is done by trying to move everything *home*. The code to do this is shown below:

```
for x in (this.contents)
  if (is_player(x))
    if (typeof(x.home) == OBJ && valid(x.home))
      x:moveto(x.home);
    endif
    if (x.location == this)
      move(x, $player_start);
    endif
  elseif (valid(x.owner))
    x:moveto(x.owner);
  endif
endfor
```

The main loop goes through the complete contents list of the room. If the object is a player, then it is moved home, using it's `:moveto` verb, or to the `$player_start` room, using the `move()` primitive. All other objects are moved to the inventories of their owners.

Note that if the attempt to move an object fails, then no action is taken. The server will place all objects still in the room into `$nothing` when the room is recycled.

`none` **@lastlog** () Verb
`none` **@lastlog** (*player*) Commabd

This verb is a player command used to list the times that players last connected to the MOO. If *player* is supplied, by invoking the verb as a command, only the last connect time for that player is shown.

If no argument is supplied, the verb uses the `players()` primitive to return a list of all players in the database. It then looks at each players `last_connect_time` property, and places a particular list, depending on whether the player conncted within the last day, week or month or longer.

When all players have been placed in one or other of the lists, they are printed out, along with the exact connect time as found in the player's `last_connect_time` property.

`none` **n*orth/e*east/w*west/s*south** () Verb
`none` **northwest/northeast/southwest/southeast** () Verb
`none` **up/down** () Verb

In general, when a player wants to move out of a room using an exit, the recognition of the exit name is done by the `:huh` and `:huh2` verbs for the room. However, to cope with the most common cases, verbs are defined for each of the compass directions and `up` and `down`. The code for these verbs is the same, and is shown below as an example:

```
exit = this:match_exit(verb);
if (valid(exit))
  exit:invoke();
elseif (exit == $failed_match)
  player:tell("You can't go that way.");
else
  player:tell("I don't know which '", verb, "' you mean.");
endif
```

What this does is check to see if the exit is defined for the room. If it is, then the exit is `:invoke`d. If not, a suitable message is sent to the player.

This case is included simply to speed up processing for certain common cases.

`none` **explain_syntax** (`string` `verb`) Verb

This verb is used to provide helpful information to the player should s/he fail to enter a command correctly. It is called by the `:huh2` verb as a last resort to process the player command. The verb follows the parser's search path for verbs looking for a match with *verb*. If one is found, this means that the parser rejected the match because the argument's did not match.

Having established this, `:explain_syntax` compares the user input to the verb argument definition, and prints some explanatory text to try and help the player enter a correct command. This verb usually catches mistakes such as entering the wrong preposition, or forgetting to use an indirect object. It is provided as part of the `$room` class to allow other room subclasses to provide more specific help for certain verbs defined in their rooms, if the user should make an error trying to use one of them.

`none` **huh2** (`list` *args*) Verb

This verb is called by the `huh` verb to handle commands that the parser couldn't sensibly pass to another object. In the case of a room, the verb covers a number of different possibilities. First, the sentence is checked to see if it matches any of the exits that lead out of the room. This is done using the `match_exit` verb. If a matching exit is found then the `:invoke` verb for the exit is called, which causes it to be activated. This provides for a flexible approach to handling exits.

If this does not produce a sensible match, then the verb is treated in one of two ways. If it starts with an 'at' (`@`) character, then we attempt to match the remainder of the verb in the sentence with a message field on the direct object. A message field is a property whose name ends in `"_msg"`. If a match is found, then this is treated as a field setting command. This approach is used to avoid having to define a verb for every message field that can exist on an object. It also allows players to add extra message fields to objects simply by ending the name of the property with `"_msg"`. For example, if you define a message on an object called `foobar_msg` then you can set the message with the command

```
@foobar <object> is <message text>
```

If the verb does not start with an 'at' (`@`) character, then we call the `:explain_syntax` verb. This tries to match the verb with a verb defined on the player, room, direct object (if any) and indirect object (if any). If a match is found, the syntax of the verb (ie, number and type of arguments and prepositions) is checked, and a useful message sent to the player.

This approach is taken to provide flexibility. By not placing this sort of code within the server, the LambdaMOO administrator has the choice of changing the way erroneous commands are handled. One application could be an augmentation of the basic `huh` action to log failed commands in a list somewhere. This mechanism, long used in other MUDs, can provide a builder with an idea of what other players have tried (and failed) to do in his or her areas.

`none` **@eject** *object* Verb

This verb is a player command used to remove unwanted objects from places that you own. It can be used to remove objects, which are sent to `$nothing` and players, which are sent to `$player_start`.

The verb first matches the supplied *object* with something in the room. If a match is found, and the player controls the current room, then the ejection messages are printed, and the victim's `:eject` verb is invoked to actually remove the object.

`string` **ejection_msg** () Verb
`string` **oejection_msg** () Verb
`string` **victim_ejection_msg** () Verb
These three verbs return the messages used by the `@eject` mechanism. The code for the verb is shown below, as it illustrates how to deal with multiple messages with a single verb:

```
set_task_perms(caller_perms());
return $string_utils:pronoun_sub(this.(verb));
```

Note the permission setting at the start of the verb. This is necessary because the verb is owned by the Wizard, and hence would have permission to read the ejection messages on any object. By restricting the permissions of the verb to those of the verb that called it, this verb restricts who may access the ejection messages on objects.

The following properties are defined for the `$room` class:

victim_ejection_msg Property
This is a string containing the message sent to objects that are ejected from a room by the owner. It is sent to the victim by the room's `:@eject` verb.

ejection_msg Property
This is a string containing the message sent to the owner when they eject an object from the room. It is printed by the room's `:@eject` verb.

oejection_msg Property
This is a string containing the message sent to everyone else in a room when the owner ejects an object. It is printed by the room's `:@eject` verb.

residents Property
This is a simple list of objects that have their homes in this room. It is examined by

the `:accept` verb on the `$room` class as one of the criteria used to determine whether an object is allowed to enter the room or not. The `@sethome` verb could be enhanced to include players as residents when their home is set to be a particular room.

free_entry Property
If this is set to a non-zero value, then the room is considered *public* and any object is allowed to move in to the room. Setting the value to zero will stop objects that do not meet any of the other acceptance criteria from entering. Refer to the `$room:accept` verb for further details on these criteria.

entrances Property
This is a simple list of the exits that lead *into* this room. It is manipulated by the `$room:add_entrance` and `$room:remove_entrance`.

blessed_object Property
This contains the object number of the object that has been *blessed* for entry into this room.

blessed_task Property
This contains the task ID of the task that requested an object be *blessed* for entry into the room.

exits Property
This is a simple list of the exit objects that lead out of this room. It is manipulated by the `$room:add_exit`, `$room:remove_exit` and `$room:match_exit` verbs.

dark Property
If set to a non-zero value, then this property inhibits display of the contents of a room when the `$room:look_self` verb is invoked. It can be used, for example, in rooms where a large number of objects are likely to be located, that you do not wish to see (for instance, a *limbo* type room, where sleeping players and lost objects reside.). If the `dark` is set to zero, then the contents of a room are displayed when `$room:look_self` is invoked.

ctype Property
This property determines the format of the room contents list, as printed by the verb `$room:look_self`. It can take a value between ‘0’ and ‘3’. An example for each of the

contents formats is given below.

`0: Traditional TinyMU* format.`

```
Contents:
  Wizard
  a newspaper
  a fruitbat zapper
```

`1: Enhanced TinyMU* format.`

```
Wizard is here.
You see a newspaper here.
You see a fruitbat zapper here.
```

`2: "All-in-one" format.`

```
You see Wizard, a newspaper, and a fruitbat zapper here.
```

`3: The default, objects and player descriptive format.`

```
You see a newspaper and a fruitbat zapper here.
Wizard is here.
```

1.1.3 The Thing Class

The `$thing` defines verbs and properties for objects that exist in the virtual world. This class includes everything that is not a player, room or exit. For example, the classes `$container` and `$note` are descended from the `$thing` class. The two basic operations that can be performed on a thing are *picking it up* and *putting it down*. Two verbs are defined for the `$thing` class to implement this idea. Configurable messages are used, so that someone using an object of this class can set the text printed when an attempt is made to take an object. No messages are available for the case of dropping an object.

`none` **g*et** () Verb
`none` **t*ake** () Verb

One or the other of these verbs is invoked when a player tries to take possession of an object i.e., pick it up. The code involved is fairly straightforward. It checks to see if the player already has the object, and prints a suitable message if this is the case. If not, then the `:moveto` verb on the object is invoked. If this results in the object moving into the player's inventory, then the `take_succeeded` messages defined on the object are printed. If the `:moveto` failed, then the `take_failed` messages for the object are printed.

This scheme allows you to add extra conditions to restrict whether a player can take an object or not. For example, you may place a notion of *strength* onto a player, and add *weight* to objects. If an object is too heavy for a player to lift, then the object cannot be taken by the player. This sort of condition should be added to the `:take` verb for the object.

`none` **d*rop** () Verb
`none` **th*row** () Verb
One or the other of these verbs is invoked when a player tries to place an object in the room s/he is currently located, i.e., when the player tries to drop the object. Again, the code is fairly straightforward. If the object is not located on the player, then a suitable message is sent to the player telling him/her to check his/her pockets. If the player does possess the object, and the current location `:accept` verb indicates that the room will allow the object to be dropped, the object `:moveto` verb is invoked to move the object from the player's inventory to the contents list of the player's location. Suitable messages are printed to inform the player that the action succeeded, and to tell other people in the room that the object has just been dropped.

`none` **moveto** (`obj` *where*) Verb
This verb should be used to move an `$thing` object from one location to another. It checks to see that *where* is a valid object, and that the lock on *wher* permits the object to enter. If this is the case, then the `$root_class:moveto` verb is invoked to actually move the object, using `pass(`*where*`)`.

`string` **take_failed_msg** () Verb
`string` **take_succeeded_msg** () Verb
`string` **otake_failed_msg** () Verb
`string` **otake_succeeded_msg** () Verb
`string` **drop_failed_msg** () Verb
`string` **drop_succeeded_msg** () Verb
`string` **odrop_failed_msg** () Verb
`string` **odrop_succeeded_msg** () Verb
These eight verbs are used to access the message fields used by a `$thing` object. In each case, the value returned by the verb is a a version of the corresponding `"_msg"` property of the object, with standard pronoun substitutions performed. This allows the messages to be more flexible than simple constant text strings. If extra complexity were to be added to the messages, then these verbs should be overridden.

The default values of these strings are shown below:

`drop_failed_msg`
: "You can't seem to drop %t here."

`drop_succeeded_msg`
: "You drop %t."

`odrop_failed_msg`
: "tries to drop %t but fails!"

`odrop_succeeded_msg`
: "drops %t."

`otake_succeeded_msg`
: "picks up %t."

`otake_failed_msg`
: ""

`take_succeeded_msg`
: "You take %t."

`take_failed_msg`
: "You can't pick that up."

The following properties are defined for the `$thing` class. They are the messages used by the `$thing:take` and `$thing:drop` verbs in various situations. All of these message can use standard pronoun substitutions.

otake_succeeded_msg Property

This message is the text sent to other people in the thing's location when the thing is taken by another object. The default value of this string is '`picks up %t.`'

otake_failed_msg Property

This message is the text sent to other people in the thing's location when an attempt by another object to take the thing fails. The default value of this string is ''

take_succeeded_msg Property

When an object successfully takes a thing, this message is sent to inform the object of success. The default value of this string is '`You take %t.`'

take_failed_msg Property

When an object unsuccessfully tries to take a thing, this message is sent to tell the object that the attempt failed. The default value of this string is '`You can't pick that up.`'

odrop_succeeded_msg Property

This message is the text sent to other people in the thing's location when the thing is dropped by another object. The default value of this string is '`drops %t.`'

odrop_failed_msg Property

This message is the text sent to other people in the thing's location when an attempt by another object to drop the thing fails. The default value of this string is '`tries to drop %t but fails!`'

drop_succeeded_msg Property

When an object successfully drops a thing, this message is sent to inform the object of success. The default value of this message is '`You drop %t.`'

drop_failed_msg Property

When an object unsuccessfully tries to drop a thing, this message is sent to tell the object that the attempt failed. The default value of this string is '`You can't seem to drop %t here.`'

1.1.4 The Exit Class

The `$exit` class is the other type of object used to construct the fabric of the virtual world. You can imagine an exit to be a flexible tube connecting two `$room` objects. Each `$exit` object goes in one direction only. It leads from a *source* object to a *destination* object. Note that it takes no virtual time to traverse an exit. When an object moves through an exit, it moves from one room to another instantaneously.

The verbs defined for the `$exit` class are fairly simple and obvious. Several messages are defined as properties on an exit. These are pronoun substituted and printed when the exit is invoked, under various conditions.

`none` **invoke** () Verb

This verb is used to trigger an exit. In the `$room` class, when a player elects to move through an exit, this verb is called to move the player through the exit. The code for this verb is very simple. It calls the exit's `:move` verb, with the `player` object as an argument.

This is the verb to use for moving players through exits. It does not allow objects to be moved. For that, a direct call to the exit's `:move` verb is needed, with the object you wish to move through the exit as an argument.

`none` **move** (`obj` *thing*) Verb

This verb is used to move *thing* through the exit. It provides a general mechanism for moving any sort of object through the exit, not necessarily just players. The code for this verb performs a number of actions. First, the lock on the exit is checked to see if *thing* is allowed to use the exit. If this is not the case, the `nogo_msg` and `onogo_msg` text is sent to *thing* and everyone else in *thing*'s location, respectively.

If the object is allowed to use the exit, it is *blessed* for entry to the destination room. This is done to ensure that the object will be accepted by the destination room. It provides a way to stop objects moving into a room by any means other than an exit leading into the room. By simply prohibiting all objects from entering the room, the only way in is then to use an exit that leads into that room.

If the object is accepted by the room, determined using the `$room:accept` verb, then the `leave` messages are printed to *thing* and the other people in the room. Then *thing*`:moveto` is invoked to move the object from the current room to the destination of the exit. Once this has been done, the `arrive` messages for the exit are printed out to *thing* and the destination room's occupants.

`none` **recycle** () Verb

This verb is defined to allow an exit to be tidily removed from the database. The exit is removed from the `entrance` and `exit` lists of the destination and source rooms respectively, if the caller of this verb has permission to do so. This is done using the `$room:remove{entrance|exit}` verbs.

`none` **leave_msg** () — Verb
`none` **oleave_msg** () — Verb
`none` **arrive_msg** () — Verb
`none` **oarrive_msg** () — Verb
`none` **nogo_msg** () — Verb
`none` **onogo_msg** — Verb

These verbs return a pronoun substituted version of the corresponding properties stored on the exit object. They are used by `$exit:move`.

The following properties are defined for the `$exit` class. Most of them are messages used when the exit is invoked.

source — Property

This is the object that the exit leads out of. If the exit has been created using the correct verbs, the exit object should be in the `exits` list of the `source` room object.

dest — Property

This is the object that the exit leads to. If the exit has been created using the correct verbs, the exit object should be in the `entrances` list of the `dest` room object.

nogo_msg — Property

This is the message printed to an object if it is unable to move through the exit.

onogo_msg — Property

This is the message printed to everyone in an object's location if it is unable to move through the exit.

arrive_msg — Property

This is the message printed to an object using an exit when it arrives in the destination room.

oarrive_msg — Property

This is the message printed in the destination room of the exit when something is moved through it.

oleave_msg Property

This is the message printed in the exit's source room when an object moves through the exit.

leave_msg Property

This is the message printed to an object using the exit as it leaves the source room.

1.1.5 The Player Class

The player class is the one of the basic classes of the virtual world. A player object is the same as any other object, except that has it's `player` bit set. This allows the server to connect a user to that object. A number of player commands are defined as verbs of the `$player` class, as well as other useful functions.

`none` **g*et** () Verb
`none` **take** () Verb

This verb is called in the situation when another object tries to *get* or *take* a player. Given that this is a fairly nonsensical thing to do, it is not allowed. Suitable messages are sent to the object that invoked the action, and the player object.

`none` **i*nventory** () Verb

This verb is used to tell a player what s/he has in his/her pockets. The code for the verb provides the best documentation, so it is printed below. Note the use of the `thing:title` verb to get the name of an object.

```
if (length(player.contents))
  player:tell("You are carrying:");
  for thing in (player.contents)
    player:tell(thing:title());
  endfor
else
  player:tell("You are empty-handed.");
endif
```

`none` **wh*isper** () Verb

This verb is used to send secret messages between two players, as if they were whispering to each other. The way this works is slightly the reverse of what might be expected, because the `:whisper` verb on the person being whispered to is the one that is invoked.

The message, referred to by the `dobjstr`, is printed to the recipient, with suitable text surrounding it to indicate that is it a whisper.

`none` **look_self** () Verb

This verb overrides the `$root_class` definition to provide an indication to other players of whether this player is currently active or not. It uses `pass()` to allow the parent class to print a description, and then looks at the `connected_players()` list to determine if this player is currently connected. If not, then the text

```
He is sleeping
```

is printed. If the player is connected, and has been idle for less than 60 seconds, then the string

```
He is awake and looks alert
```

is printed. If the player is connected, but has been inactive for more than 60 seconds, the string

```
He is awake, but has been staring off into space for X
```

is printed, where `X` is an indication of the time the player has been inactive. The gender pronoun inserted is taken from the pronoun list for the player. This means it can vary with the gender of the player object.

If the player is carrying any objects, a simple list of these is printed.

`none` **@gag** (*args*) Verb

The concept of gagging is fairly simple to understand. If there is a player or object you do not wish to see any output from, then you can place the player or object into your *gag list*. If any output is sent to you from that object, using it's `:tell` verb, then it is ignored, and not printed on your terminal screen.

Two gaglists are maintained: one for players, in the property `gaglist` and one for objects, in the property `object_gaglist`.

Three verbs are used to maintain and observe the list of objects that are in the gag lists. The `@gag` verb is used as a player command to add objects to the gag lists. The code for this is fairly straightforward, and is included below:

```
if (player != this)
  player:tell("Permission denied.");
  return;
endif
victims = $string_utils:match_player(args);
if (!victims)
  player:tell("Usage:  @gag <player or object>
                     [<player or object>...]");
  return;
endif
gagplayers = gagobjs = {};
for i in [1..length(args)]
  if (valid(victims[i]))
    gagplayers = {victims[i], @gagplayers};
  elseif (valid(o = player.location:match(args[i])))
    gagobjs = {o, @gagobjs};
  elseif (tonum(o = toobj(args[i])) && valid(o))
    gagobjs = {o, @gagobjs};
  else
    player:tell("Could not find ", args[i], " as either a player or
an object.");
  endif
endfor
changed = 0;
for p in (gagplayers)
  if (p in player.gaglist)
    player:tell("You are already gagging ", p.name, ".");
  else
    changed = 1;
    player.gaglist = setadd(this.gaglist, p);
  endif
endfor
for o in (gagobjs)
  if (o in player.object_gaglist)
    player:tell("You are already gagging ", o.name, ".");
  else
    changed = 1;
    player.object_gaglist = setadd(this.object_gaglist, o);
  endif
endfor
if (changed)
  fork (0)
    this:("@listgag")();
  endfork
endif
```

Note that you can only manipulate your own gag list. Other objects cannot change your gag list. Another interesting point is the call to `@listgag`, after the new objects have been added to the gag list, to print the current gag list.

The code first tries to match the argument list against the players in the database, using `$string_utils:match_player()`. Then, for each argument given, if a match was not found using that method, a match for an object in the room is tried, using the utility routine `$string_utils:match_object()`. If this fails, the argument is rejected.

This results in a list of objects to be added to the player's gag lists. Any objects already in the player's gag lists are removed from the *to do* list. If this yields an empty *to do* list, the command is aborted. Any objects that are not already in the gag list are added to it.

Finally, `this:@listgag` is called to print the list, if it has changed.

`none` **@listgag** () Verb

This verb is used as a player command to list the objects in the player's gag lists. If the gag lists are not empty, a list of the names of the players in the `gaglist` property on the player, and the objects in the `object_gaglist` property are produced.

The verb uses a test based on the `callers()` primitive, to determine if it is being called by another verb, or invoked by a player as a command. If `callers()` returns a null list (and hence is not `TRUE`, then the verb is being invoked as a player command. This affects the text printed by the verb.

In addition, the verb checks through the gag lists of all the player's in the database, to see if the player who invoked the command is listed in anyone else's gag list. If this is the case, a list of the people doing the gagging is printed.

`none` **@ungag** (*args*) Verb

This verb is used as a player command to remove an object from one of the player's gag lists, if it is a member. The `dobjstr` is used as the name of the thing to remove from the gag list.

If this name is '`everyone`' then the player gag and object gag lists are reset to the empty list. Otherwise, if a valid direct object has been referred to, by checking `dobj`, that is used as the object to gag. Otherwise, an attempt is made to match the `dobjstr` with something in the player gag list. If no match is found, it is retried with the object gag list. If this fails, the command is aborted with an error message.

If a valid match is found, it is removed from the relevant list, and the player informed. `@listgag` is used to print the list after the operation has been completed.

`none` **news** () Verb

This is a very simple verb used as a player command to print out the current MOO news. It makes a simple call to the verb `$news:read` to actually perform the reading task.

`none` **`@`gripe** (*args*) Verb

A *gripe* is a player complaint or observation that is sent, using MOO mail, to all the administrators in the game. It is intended to provide a way to report problems with the MOO in a high-priority way that will attract the attention of the people who can do something about it. A good example of the use of a gripe is to complain about a bug in one of the core classes, or maybe even a request for something to be added.

The implementation of the gripe concept involves a property on the *System Object* called `$gripe_recipients`. This is a list of all the players who will be mailed the `@gripe` message. When a player types in `@gripe` they are taken to the mail room to enter their message. Any text entered on the line with the `@gripe` command is taken to be the subject of the gripe message. When the message is finished and sent, it is received by all the people on the `$gripe_recipients` list.

`none` **gi*ve** () Verb
`none` **ha*nd** () Verb

This verb is a player command used to exchange objects between players. It performs a `:moveto` on the direct object to the inventory of the indirect object. If, after the move, the object is still in the possession of the donor, then it is obvious that the recipient has rejected the gift. If this is the case, a suitable message is printed to the doner.

`none` **?** `help info*rmation` () Verb

This verb is a player command used to activate the help system. It is described more fully in the section on the help system.

`none` **`@`password** () Verb

This verb is a player command used to change a player's password. A player's password is stored in a property on the player called, surprisingly enough, `password`. The `crypt()` primitive is used to store the password in encrypted form, using the standard UN*X encryption algorithm. Note that you need to know your old password in order to change it, unless a password has not been previously set for this player.

`none` **@gender** (`string` *gender*) Verb

This verb is used to set the gender of a player object. Without any arguments, it prints the player's gender, currently set pronouns and the available genders stored on `$gender_utils.genders`.

If a gender is given as the argument, *gender*, then the `$gender_utils:set` verb is called to actually change the player's gender pronouns. If this verb does not return an error, then the gender of the player is set to the full gender name which is returned. `$gender_utils:set` takes care of setting the correct pronouns.

If an error is returned when trying to set the player's gender, this could indicate that permission to change the pronouns was denied, or some other problem existed. If a value of `E_NONE` is returned by `$gender_utils:set` then the gender of the player is set, but the pronouns are left unchanged.

The gender of a player is used in the `string_utils:pronoun_sub` verb to insert the appropriate pronouns in places where '%' substitutions have been used. When the gender of a player is changed, it results in a set of 8 properties being assigned on the player, one for every type of possible pronoun substitution. A further property, containing the gender of the player, is also set, for example, to either "male", "female", or "neuter", depending on the argument given to the `@gender` command.

`none` **page** (`list` *args*) Verb

This verb is a player command used to send messages between players who are not physically located in the same room in the virtual world. You can imagine a *page* to be a worldwide form of whispering. Without an argument, a message like

```
You sense that blip is looking for you in The Venue Hallway.
```

is sent to the recipient of the page. If an argument is given, it is treated as a message to send to the other player. This results in the recipient getting a message like

```
You sense that blip is looking for you in Hallway.
He pages, "Hello - are you busy ?"
```

Paging is used primarily to attract the attention of a player, or to pass short messages between players in different locations. It is not intended to be used for conversation.

If a player name has been given as an argument, the `:page` verb first tries to match the first argument with a player name, using `$string_utils:match_player`. If a match is found, then there are two possibilities. Firstly, if the player is not connected, a pronoun substituted version of the string returned by that player's `:page_absent_msg` verb is printed to the sender, and the verb is aborted.

Otherwise, if the recipient is connected, we send him/her the string returned by the sender's `:page_origin_msg`. We then check to see if, optonally, '`with`' followed by a message is part of the argument list for the verb. If so, then the message is extracted from the argument list and sent to the recipient, suitably pronoun substituted. The string returned by the recipient's `:page_echo_msg` verb is printed to the sending player.

An interesting piece of coding is used to stop the line containing the message from duplicating the sender's name if it has already been sent as part of the `:page_origin_msg`. For example, if blip page's Ezeke, Ezeke might see the following:

```
You sense that blip is looking for you in The Venue Hallway
He pages, "Hello"
```

which would be better than something like

```
You sense that blip is looking for you in The Venue Hallway
blip pages, "Hello"
```

The code in question is shown below:

```
who:tell($string_utils:index_delimited(pagemsg, player.name) ?
          player.psc | player.name, " pages, \"", msg, "\"");
```

Here, the `$string_utils:index_delimited()` verb is used to check if the player's name occurs in the string we sent to the recipient as `pagemsg`. If it does, then we print the player's subjective pronoun, capitalised. If it doesn't, we print the player's name.

`none @rename` () Verb

This verb is a player command used to change the name and aliases of an object or verb. It first tries to parse the supplied direct object as a verb reference. If this succeeds, then it tries to match the object, and then the verb name. If both are found, the indirect object string is used to set verb name.

If the direct object string cannot be parsed as a verb, then a match is attempted using the string as an object name. If this succeeds, then `$building_utils:set_names()` is used to set the name and aliases of the matched object.

Any permission errors, ambiguous object specifications or syntax errors are flagged appropriately.

`num` **accept** (`obj` *thing*) Verb

This verb is used to determine if a player will allow *thing* to be moved into his/her inventory. The verb defined for the `$player` class allows anything that is not a player to be moved into the player's possession. You could override this verb to restrict the sorts of things you would want other people to be able to place on your person.

`none` **tell** () Verb

This verb should be used to send a message to a player. The `$player:tell` filters messages in two different ways, as show below. Remember that the *player* referred to in the code is the player sending the message. *this* refers to the player receiving the message.

```
if (!this:gag_p())
  if (this.paranoid == 2)
    z = this:whodunnit(listappend(callers(), {player, "", player}),
              {this}, {})[3];
    pass("(", z.name, " ", z, ") ", @args);
  else
    pass(@args);
    if (this.paranoid == 1)
      this.responsible = listappend(this.responsible,
          {listappend(callers(), {player, "<cmd-line>",
                             player}), args});
      while (length(this.responsible) > this.lines)
        this.responsible = listdelete(this.responsible, 1);
      endwhile
    else
      this.responsible = {};
    endif
  endif
endif
```

The verb `$player:gag_p` returns true if the player sending the message is in the recipient's *gag list*. For this verb, the output from any gagged player is ignored, and not printed to the recipient's terminal.

If the *paranoid level* of the recipient is '`2`', this means that they wish to see who has sent them a message. The `$player:whodunnit` verb returns the object reference of the player that sent the message. This is prepended to the message text, which is then printed to the player.

If the *paranoid level* of the recipient is '`1`', then the message and it's originator are stored in the property list `responsible` on the player. The list is kept to *player.lines* length, at most. This option is used for later processing by the `@check` command.

`list` **whodunnit** (`list` *callers*, `list` *trust*, `list` *mistrust*) Verb

This verb runs through the list of *callers* until it finds an object reference that is not a wizard, is not in the list *trust*, or is in the list *mistrust*. The verb is used by `$player:tell` to locate the originator of a message. It returns a list of three elements, in the same format as that returned by the `callers()` primitive:

{*this*, *verb-name*, *programmer*}

where *this* is the initial value of the variable `this` in that verb, *verb-name* is the name used to invoke the chosen verb, and *programmer* is the player with whose permissions that verb is running.

`num` **gag_p** () Verb

This verb returns '`TRUE`' if the current value of the `player` variable is in the gag list of the object indicated by the variable `this`, or if a non-player object mentioned in the gag list if the first elements of the verbs `callers()` list.

`none` **home** () Verb

This verb is normally invoked by the player as a command. The *home* of a player is the room where s/he goes to when they disconnect. Unlike some flavours of MUD, MOO does not cause you to lose all your possessions if you go home.

The `:home` verb performs a simple sequence. It first checks whether the player is already at home, and tells him/her so if this is the case. Secondly, a check is made that the player's home (stored in the `home` property on the player, is a valid object. If this is not the case, the verb sets the home to the default, `$player_start`, and stops with a suitable message.

Having decided that the player has a valid home s/he is not already in, the verb uses `$player:moveto` to send the player to the home location. If this doesn't work - checked by comparing the player's home with the player's location after the move - then for some reason the player's home location has not allowed him/her to enter. A suitable message is printed, and no further action is taken.

`none` **@sethome** () Verb

This verb is a player command used to set the player's *home* to be the his/her current location. You can only set your home to a room if the room will accept you, determined by checking the room's `:accept` verb.. This allows builders to restrict which rooms may be used by players as their homes. If the room does not allow the player to set it as his/her home, a suitable message is printed to inform the user of this fact. Otherwise, the player's `home` property is set to the player's current location.

`none` **confunc** () Verb

This verb is called when the player connects to the LambdaMOO server. It can be used to perform actions that should be done every time the player logs in to the MUD. This is much the same idea as having a `~/.cshrc` or `~/.kshrc` file that is activated when you log into a UN*X account. The verb defined for the player class is listed below:

```
$news:check(this);
for x in (this.messages)
  if (x[1] > this.current_message)
    this:tell("You have new mail.  Type 'help mail' for
                                  info on reading it.");
    return;
  endif
endfor
```

This performs a couple of actions. First it calls `$news:check` to see whether the news has been updated since this player last looked at it. Then it checks through the MOO Mail list on the player to see if any mail has arrived since they were last connected.

You could place a variety of actions into this verb. For example, you may wish to tell your friends when you log in, by sending a suitable message to them if they are connected. Similarly, you may wish to produce a special message in the room you are in when you connect.

none disfunc () Verb

This verb is invoked when the player disconnects from the LambdaMOO server. The verb defined for the `$player` class does nothing. You might choose to override this verb to print a special message when you log out, or perhaps to retrieve some of your possessions.

none whereis (*args*) Verb

none @whereis (*args*) Verb

This verb is a player command used to locate a player in the virtual world. The code, shown below, allows any player to locate any other player. If no argument is given, a list of the locations of all the currently connected players is printed.

```
if (!args)
  them = connected_players();
else
  who = $command_utils:player_match_result(
                      $string_utils:match_player(args), args);
  if (length(who) <= 1)
    if (!who[1])
      player:tell("Where is who?");
    endif
    return;
  elseif (who[1])
    player:tell("");
  endif
  them = listdelete(who, 1);
endif
lmax = rmax = 0;
for p in (them)
  player:tell(p:whereis_location_msg());
endfor
```

If an argument is given, the verb attempts to match with one or more player names. If no valid matches are found, a suitable error message is printed by invoking the verb `$command_utils:player_match_result`. That verb returns a list. The first element indicates whether any of the elements of the argument list didn't match. The rest of the list contains the objects references that did match.

The verb runs through the list of object references, and reports the string returned by each player's `whereis_location_msg` verb.

`none` **@mail** () Verb
`none` **@rmm*ail** () Verb
`none` **@send** () Verb
`none` **@read** () Verb
`none` **@answer** () Verb
`none` **@reply** () Verb
`none` **@renumber** () Verb
`none` **receive_message** () Verb
`none` **fix_current_message** () Verb
`none` **@next** () Verb
`none` **@prev** () Verb

These functions are player commands used to access the MOO Mail system. They are fully described in the section on the MOO mail system.

`none` **@quota** () Verb

This verb is a player command used to print a player's current building quota. This is a numerical value of the number of objects that the player may create without recycling anything. Quotas are used to limit the number of objects that can be created, in principle to allow the game administrators to quality control the things that are placed in the database.

With no argument, the player's quota is displayed, taken from the property `ownership quota`. If an argument is given in the `dobjst`, it is taken as a player name, and matched to find a player object reference. If one is found, and the user is a wizard, then the value of that player's `ownership_quota` is returned. Otherwise, a '`permission denied`' message is returned.

`none` **@realm** (*args*) Verb

This is a player command used to examine the object hierarchy. The arguments can specify the players whose objects are to be shown, the root object number to start from, and a list of objects to miss from the tree search.

Firstly, the verb parses the arguments, to determine the root to start from, the players whose objects we are intersted in, and any objects that are to be ignored. Any ancestors of the root are printed by a short loop, with the owner appended in square brackets. The verb `this:realm2` is called to produce the hierarchical list of objects to printed. A summary of the number of objects in the list, and how many are owned by the player, is printed at the end.

`list` **realm2** (`obj` *root*, `list` *owners*, `string` *space*, `list` *missing*) Verb

This verb puts toegther a number of lines of text indicating the object hierarchy from the *root* object downwards. Only objects owned by people in the *owners* list are included. Objects in the *missing* list are excluded. The verb is called recursively to evaluate the descendants of each node in the tree of objects. The string *space* is the visual indentation factor used to represent different levels in the hierarchy.

`none` **@count** () Verb

This verb is a player command used to print the number of objects owned by a particular player. If no player name is given in the `dobjstr`, then the number of objects owned by the player invoking the verb is printed. In both cases, the verb `this:count_2` is called to do the actual counting and displaying.

`none` **count_2** (`num` *count*, `num` *start*) Verb

This verb is called to count the number of objects owned by the player indicated by `dobj`. The verb first checks the current remaining time available for it to execute in. If it is in danger of running out of time, it forks another `:count_2` verb to carry on from where it left off. This enables the verb to count lots of objects in a large database.

Foe each object, if it is valid, and the owner is the same as `dobj`, then the count is incremented. This is performed for every object in the database, from *start* to the last item.

When all the objects in the database have been looked at, the total is printed.

`num` **set_name** () Verb

This verb is used to set the name of the player object. It overrides the verb `$root_class:set_name` to provide some extra checking needed for player objects. This checking basically stops you changing your name to be the same as another player. Note also that spaces are not allowed in player names. If the extra checks are passed, the parent `set_name` verb is called to actually set the name.

`none` **@memory** () Verb

This is a player command used to return memory usage statistics for the LambdaMOO server. It uses the `memory_usage()` primitive to gather usage information, which is then printed in a nicely formatted fashion.

`none` **announce** (*args*) Verb
`none` **announce_all_but** (*args*) Verb

This verb is used to call the `:announce` and `:announce_all_but` verbs in the player's location, if those verbs are defined, and the player is in a valid location. It is used by noisy objects in player's inventories that wish to broadcast messages to both the player, and others in the surrounding room.

`none` **@notedit** (*args*) Verb

This is a simple verb used as a player command to invoke the note editor on the argument.

`none` **@who** (`list` *names*) Verb

This verb is a player command used to determine who is currently connected to the MOO. If *names* is given, it is a list of player names to display in the *who list*. In both cases, the list is sorted, using `$list_utils:sort_alist` and displayed. Player names, connection times, idle times and locations are displayed. Player name is determined by getting the value of the `name` property on the player. Connection time is determined by using the `connected seconds()` primitive. Idle time is determined using the `idle_seconds()` primitive. The location is determined by the return value of a player's `:who_location_msg` verb.

A summary of player activity is attached to the end of the list.

`none` **@typo** (*args*) Verb
`none` **@bug** (*args*) Verb
`none` **@suggest*ion** (*args*) Verb
`none` **@idea** (*args*) Verb

These commands are used register a typo/bug/suggestion/idea with the person who owns the room the player is currently in. If *args* is given, this is used as the text of the message, and is delivered straightaway to the room owner.

If no argument is given, the player is taken to the mail room to compose a full message. However, if the room the player is in does not have it's `free_entry` property set, then the verb doesn't move the player to the mail room, as s/he may not be able to reenter the current room after the message has been completed. Similarly, if the `$generic editor` is a parent of the current room, then the move is not performed. Generally speaking is is not a good idea to be taken to the mail room from within another editor.

`none` **@quit** () Verb

This verb is a player command used to exit in an orderly fashion from the MOO server. It simply calls the `boot_player` primitive to disconnect the player.

`none` **@audit** () Verb

This verb is a player command used to print the objects owned by this or another player. If a direct object is given, this is taken as a player name to audit, otherwise the current value of `player` is used. In both cases, the `this:audit_2` verb is called to do the actual counting and displaying of objects.

`none` **audit_2** (`num` *count*, `num` *start*) Verb

This verb is called by the `@audit` command to print the objects owned by player given in `dobj`. In a similar way to `@count`, the verb first checks how much time it has left. If it might run out, then it forks another occurrence of itself to carry on auditing where it left off.

A loop is entered, running from *start* to the last item in the database. Each object in the database is looked at. If it is valid, and the owner matches the `dobj`, then the object name and reference number are printed, and the count is incremented.

When all the objects in the database have been covered, the total count is printed out.

`none` **@eject** () Verb

This verb is a player command used to remove objects from rooms or the player's inventory. The indirect object is used to specify what the direct object is to be ejected from. The special cases of '`me`' and '`here`' are caught and handled. Similarly, the case where the direct object is '`me`' is also handled.

If neither of the special cases match, the direct and indirect object are matched on, to give an object reference to use. Suitable error messages are printed if a match is not found. If matches are found, then a sequence of message printing is started. The indirect object's `:victim_ejection_msg` verb is invoked, and the returned result printed to the victim. The string returned by the indirect object's `:ejection_msg` is printed to the player doing the ejecting. The result returned by the indirect object's `oejection_msg` verb is printed to everyone else in the room.

Finally, the indirect object's `:eject` verb is called to remove the victim.

`none` **QUIT** () Verb

This player command used to be the way to disconnect from the MOO. It has been replaced with the '`@quit`' command. Some rooms, for example editors, define the '`QUIT`' command. If the room the player is in has a `:QUIT` verb, it is called. Otherwise, the verb prints a message telling the player to use the '`@quit`' command.

`none` **@unlock** () Verb

This verb is a player command used to unlock an object. The direct object string, `dobjstr` is matched to try and find an object to unlock. If a match is found, the `key` property is reset to '`0`'. Any errors are reported to the invoking player.

`none` **@lock** () Verb

This verb is a player command used to lock an object with a specified key. It first matches the direct object, `dobj` to get an object reference number. If that succeeds, the `$lock_utils:parse_keyexp()` verb is called to parse the key expression given for thelock. If that fails, a suitable error message is printed. Othwerwise, the `key` property of the object being locked is set to the returned value from the parsing verb. Again, any errors are reported to the invoking player.

`none` **@version** () Verb

This is a player command used to print the current server version number, using the `server_version()` primitive.

`none` **@mess*ages** () Verb

This is a player command used to list the possible message properties (those that end in '`_msg`') on the `dobjstr` supplied. The names of the messages, along with their current values, are displayed.

`none` **@sw*eep** () Verb

This is a player command used to list the objects that are capable of receiving any messages sent out from the player in the current room. It gathers a list of the objects in the current room, from the `contents` property of the player's location. In any element in the list is a connected player, the message

```
blip (#42) is listening.
```

is printed, for example.

If an element has a `sweep_msg` verb defined, the returned string from this verb is printed, prepended by the object's name.

If an element has a `tell` verb defined, and the owner is not the invoking player or a wizard, then this object is a potential snooper, and is reported by a phrase such as:

```
The Fruitbat (#999) has been taught to listen by blip (#42).
```

The verbs '`announce`', '`announce_all`', '`announce_all_but`', '`say`', '`emote`', '`huh`', '`huh2`' and '`whisper`' are checked to see if the current room has a definition not owned by the invoking player or a wizard. If any of these are found, a message such as:

```
The Venue Hallway (#1234) may have been bugged by blip.
```

if the player's location was the Venue hallway.

If no potential bugs are found, the the message '`Communications are secure.`' is printed, and the player can breath easily (ish).

`none` **`@che*ck`** (`list` *args*) Verb

This is a player command used to check the origin of the last few messages received by the player. The *args* list can contain the number of lines to print as the first element, followed by a list of player names. Each player name in the list is a person to be *trusted*, unless the name is prefixed by an exclamation point, '`!`', in which case the person is not to be trusted.

The verb starts by building up a list of trusted and mistrusted individuals based on the names given on the command line. Then it runs through last *n* messages in the player's `responsible` property list, checking the origin of the messages using the `this:whodunnit` verb with the *trust* and *mistrust* lists.

Any dubious messages found are printed, along with details of who sent them.

`none` **`@move`** () Verb

This is a player command used to move objects from one location to another. The direct and indirect object strings are used to match an object and location to move to. The `:moveto` verb on the `dobj` is invoked to move it to the `iobj` location. If the location changes, and the object is a player, then the message

```
blip disappears suddenly for parts unknown.
```

is displayed in the player's location before the move took place. Similarly, at the new location, the message

```
blip materializes out of thin air.
```

is printed to the objects in the destination room. No messages are printed if an object is moved. If a permission problem is found, it is reported to the player who typed the command.

`none` **@par*anoid** () Verb

This verb is used to set the player's *paranoid level*, as stored in the `.paranoid` property. Three different levels are available :

- 0. The normal case, where no paranoia is applied to any messages sent to the player.
- 1. In this case, the anti-spoofer is enabled, and the value of the `lines` property on the player is set to 20. This determines how many messages are stored on the player, for checking with the '`@check`' command.
- 2. In this case, every message sent to the player is prefixed with the name and object number of the sender. This is the *immediate* mode of the anti-spoofer mechanism.

`none` **@s*how** () Verb

This is a player command used to examine other objects in detail. It returns details of an object's owner, location, parent, and all the verbs and properties defined for the object, along with their values, if permissions allow.

`none` **@desc*ribe** () Verb

This is a player command used to set the description of an object. It takes the `dobjstr` and tries to match it with an object. If a match is found, then the object's `:describe` verb is invoked, with the `dobjstr` as an argument.

`none` **@create** (*args*) Verb

This verb is a player command used to create an instance of an object or class. It parses the arguments to extract the parent object we wish to create an instance of, along with the name we wish to give to the created object. The `create()` primitive is called, with the derived *parent* as an argument. The resulting object is moved to the player's inventory, using the `move()` primitive, and it's names and aliases set using `$building_utils:set_names`.

none @recycle () Verb

This is a player command used to recycle an object. This simply matches the `dobjstr` with an object, and calls the `recycle()` primitive to recycle the object. The returned value, in the case of an error, is printed to the player. Otherwise, a suitable success message is sent.

none @dig () Verb

This is a player command used to create a room or exit, (that is, instances of the class `$room` or '`$exit`'. The verb parses the arguments to determine the type and number of objects that are to be created. It uses the `create()` primitive, with `$room` as a parent to create a room. Note that you can only use the `@dig` command to dig an exit from within a room. The `building_utils:make_exit` verb is called to dig an exit.

none @cl*asses () Verb

This is a player command used to list the publicly available classes in the database. The public classes are divided into different sections, as defined by the property `$class_registry`. This contains a list of entries, in the following form:

```
{ {name, description, members}, ... }
```

where *name* is the name of the superclass of objects, *description* is a short, one line description of the superclass, and *members* is a list of object references that are in the superclass.

If no argument is given to the '`@classes`' command, it runs through the entries in the `$class_registry` and prints out the name and description.

If an argument is given, it is taken as one or more names of an entries in the `$class_registry`. The names of the *members* of the requested classes are printed, in this case, using `this:classes_2`.

none classes_2 (`obj` *root*, *indent*, `list` *members*, `list` *printed*) Verb

This verb is called by '`@classes`' to print the members of the classes given in *members*.

It is called recursively to list all the children of every class in *members*. If *root* is in the list of *members* then it is printed as the class root name. Otherwise, it is printed as a member of the class. For every child of the *root*, if it is in the set *printed*, `this:classes_2` is called, to print any descendants.

`str` **who_location_msg** () Verb
`str` **whereis_location_msg** () Verb
`str` **page_origin_msg** () Verb
`str` **page_echo_msg** () Verb
`str` **page_absent_msg** () Verb
These verbs return the value of the corresponding property on the player after pronoun substitution. They are provided to allow the player to override the verbs for more distinctive and complex messages.

The following properties are defined for the `$player` class.

current_message Property
This property contains an integer index into the list of MOO Mail messages currently stored in the `messages` property. It is used by the mail handler to determine which message the player is currently looking at.

messages Property
This property contains a list of mail messages. Each mail message is a list in the following format:

`{`*num*`, {`*line-1*`, `*line-2*`, ..., `*line-n*`}}`

num gives the message number used in the various MOO mail commands. The list of strings contains the text of the message, one line per string.

last_connect_time Property
This property is an integer, holding the time that this player last connected to the MUD. The integer is in the format returned by the `time()` primitive, i.e., the number of seconds elasped since 1 January 1970, GMT.

ownership_quota Property
This property stores the current quota for the player. It is not accessible by the player,

to prevent you increasing your building quota arbitrarily. This field can be only changed by a wizard.

gender Property

This property is used to hold a string representing the gender of the player. In the normal case, this is one of '`male`', '`female`', '`neuter`'. This property is affected by the '`@gender`' command.

ps Property

This property is used to hold the subjective pronoun, set to either "he", "she" or "it" as appropriate. `string_utils:pronoun_sub` replaces the string `"%s"` with the value of this property.

po Property

This property is used to hold the objective pronoun, set to either "him", "her" or "it" as appropriate. `string_utils:pronoun_sub` replaces the string `"%o"` with the value of this property.

pp Property

This property is used to hold the possessive pronoun, set to either "his", "her" or "its" as appropriate. `string_utils:pronoun_sub` replaces the string `"%p"` with the value of this property.

pr Property

This property is used to hold the reflexive pronoun, set to either "himself", "herself" or "itself" as appropriate. `string_utils:pronoun_sub` replaces the string `"%r"` with the value of this property.

pq Property

This property is used to hold the possessive pronoun in it's noun form, set to either "his", "hers" or "its" as appropriate. `string_utils:pronoun_sub` replaces the string `"%q"` with the value of this property.

psc Property

This property is used to hold the subjective pronoun, set to either "He", "She" or "It" as appropriate. `string_utils:pronoun_sub` replaces the string `"%S"` with the value of this property.

poc Property

This property is used to hold the objective pronoun, set to either "Him", "Her" or "It" as appropriate. `string_utils:pronoun_sub` replaces the string `"%O"` with the value of this property.

ppc Property

This property is used to hold the possessive pronoun, set to either "His", "Her" or "Its" as appropriate. `string_utils:pronoun_sub` replaces the string `"%P"` with the value of this property.

prc Property

This property is used to hold the reflexive pronoun, set to either "Himself", "Herself" or "Itself" as appropriate. `string_utils:pronoun_sub` replaces the string `"%R"` with the value of this property.

pqc Property

This property is used to hold the possessive pronoun in it's noun form, set to either "His", "Hers" or "Its" as appropriate. `string_utils:pronoun_sub` replaces the string `"%Q"` with the value of this property.

home Property

This property stores the object number of the player's *home*. This is the place that the player is taken to when s/he types in '`home`'. The home room is normally either a general sleeping area, or the player's hotel room or apartment.

password Property

This property stores the encrypted password used to connect to the LambdaMOO server as this player. It is changed by using the '`@password`' command, and is checked by the server upon login.

gaglist Property

This property stores a list of object numbers. Any output received, using `$player:tell` from any of the objects in the list is filtered out of the text sent to the player's terminal screen. Items are added and deleted to this list using the '`@gag`' and '`@ungag`' commands.

paranoid Property

This property is an integer, representing the *paranoid level* of the player. If set to '`2`', any output received by the player, via `$player:tell` is prefixed with the name of the object that originated the message. This is used to that a player can detect spoof output from other players or objects.

If set to '`1`', any messages sent using `$player:tell` are stored in a small cache in the property `responsible`, along with details of where the message came from. This can be used for later checking with the '`@check`' command.

If set to '`0`', no spoof checking is performed. This is the normal case.

responsible Property

This property stores a list in the following format:

```
{callers, text}
```

The *callers* part is in the same format returned by the `callers()` primitive, and lists the *verb stack* at the time that the message stored in *text* was sent.

This property is set by `$player:tell` and looked at by `$player:@check`. It contains a maximum number of entries, as determined by the value of the `lines` property on the player.

lines Property

This property stores the number of lines kept in the property `responsible` for anti-spoof checking.

object_gaglist Property

This property holds a list of objects that are in the player's *gaglist*. Any output from these objects is ignored by the player's `:tell` verb.

page_absent_msg Property

This property holds the message sent to someone when they try to page the player while s/he is not connected. It may contain the usual pronoun substitutions. The default value is '`%n is not currently logged in.`'

page_origin_msg Property

This property holds the message sent to a player you page, telling them where you are. The default message is '`You sense that %n is looking for you in %l.`'

page_echo_msg Property

This property holds the message sent to a player that pages you when you are logged in. The default message is '`Your message has been sent.`'

whereis_location_msg Property

This property holds the message sent when someone locates you through the '`@whereis`' command. The default message is '`%N (%#) is in %l (%[#l]).`' This might expand out to something like

```
Blip (#35) is in The Toilet (#96).
```

who_location_msg Property

This property holds the message printed in a *who list* to inform other player's of your location. The default value is '`%l`', which returns your current location.

mail_notify Property
mail_options Property
mail_forward Property

These properties define your MOO mail environment. They are discussed in more detail in the section on the MOO Mail system.

edit_options Property

This property defines your MOO editor environment. It is discussed in more detail in the section on the MOO editor.

1.2 The Mail Room Class

*** To be updated for MOO 1.3 ***

This class provides the mail facility used within LambdaMOO. Some would say that to provide an electronic mail facility within a game is slightly insane, but it is very useful for passing around game related messages. Coupled with the use of a suitable client program, for example the excellent `mud.el` with MOO Mail extensions, sending MOO mail can be easy and efficient. It also allows players to separate their MOO based mail from other, more important mail.

LambdaMOO does not yet provide any facility for reading input from a player. For this reason, programs such as the mail room use a pseudo-room to catch player input. For example, when you start to compose a mail message, you are transported to the mail pseudo-room. Once there, anything you enter is trapped by various verbs in the room, and stored in the message you are entering.

This method means that commands used in the mail composition environment are actually verbs on the mail pseudo-room.

Mail messages are actually stored on the player object, in a property list called `messages`. Each message occupies one entry, called a *mail record*, in the following format:

```
{num, {date, from, to, subject, line1, ..., lineN} }
```

Apart from the *num* field, these are all strings. The *num* field stores the message number for each message. When a new piece of mail arrives, it is allocated a message number 1 greater than the last message in the player's mailbox. If the mailbox is empty, the message is numbered '`1`'. Each message retains it's unique message number until it is deleted using '`@rmmail`'.

The number of messages in the mailbox is given by the length of the messages list. A *current message* pointer is maintained in the property `$player.current_message`. This is used to determine which message to print when the '`@read`' command is used.

The MOO Mail system is implemented by four separate objects, and some verbs on the `$player` class:

- The Mail Room/Editor `($mail_room or $mail_editor)`. This room is the place where mail messages are composed. It also contains a number of mail utility verbs. The mail editor is a child of the `Generic Editor`.
- The Mail Distribution Center `($mail_agent)`. This is a child of `$root_class` and has verbs used to handle mail distribution. It is also the database of *mailing lists* and *mail folders*.
- The Generic Mail Recipient `($mail_recipient)`. This is a special class used to set up *mailing lists* and *folders*.
- The Player Class `($player)`. This class holds various player commands used to deal with the MOO Mail system.

The sections that follow detail the verbs in each of the classes, along with any relevant properties.

1.2.1 Mail Commands and Properties on The Player Class

A number of verbs are defined in the `$player` class to provide access to the mail system. These verbs are detailed below:

`none` `@mail` (*args*) Verb

This command is used to list the messages currently stored in the player's mailbox. Mail messages are stored in the property list `messages` on the player. This is a list of complete messages, stored one per list item, in the format shown above.

The '`@mail`' command first checks the current message number still exists, using the verb `this:fix_current_message`. It then attempts to parse the arguments given to it, to get a set of message numbers that the user wishes to have displayed. The possible arguments are as follows :-

`cur`	- the current message
`new`	- all messages after the current message
num	- (where *num* is a number) the message numbered *num* if there is one.
num1-num2	- all messages in the given range, if any.
`last:`*num*	- the last *num* messages
`-`*num*	- the last *num* messages

These may be combined on the '`@mail`' command line. When a list of messages to display has been gathered in the variable `msglist`, the following code is used to display the list to the user:

```
for x in (msglist)
  msg_num = x[1];
  player:tell(this.current_message == msg_num ? "-> " | "   ",
              $string_utils:right(msg_num, 2), ":  ",
        $mail_recipient:summary_line(x[2]));
    endfor
```

For example,

```
    You have 3 messages:
        1:  May 30 12:18   Ezeke (#57)   Classes Documentation
        2:  Jun  4 19:35   Ezeke (#57)   Room contents using title()
    ->  3:  Jun  5 10:23   Ezeke (#57)   Potential problems
```

The pointer, '`->`' indicates the current message. The date, subject and sender of each message, along with the message number, is taken from the relevant mail record stored in the player's `messages` property.

`none` **@rmm*ail** (*args*) Verb

This command is used to delete mail from the player's mailbox property, `messages`. The verb takes each message number given as an argument, and searches for the message in the player's mailbox. If an argument is the string '`cur`', then this is taken as the number of the current message.

If a match is found, by checking the first entry in each *mail record*, the message is removed from the `messages` property list. If a message is not found, the player is informed with a suitable message.

`none` **@send** *player1 player2 ...* Verb

This command is used to send mail messages to other players. The verb simply calls

```
    $mail_editor:invoke(args, verb, this.mail_options)
```

to start composition of a message.

`none` **@read** Verb

`none` **@read** *num1 num2 ...* Verb

This command is used to read mail messages in the player's mailbox. If no argument is given, the *current message* is displayed on the player's terminal screen. If a list of one or more message numbers is given, each message is printed out, in turn.

The verb first checks to see if an argument has been given. If this is not the case, it attempts to get the current message number, and use this as the argument. If there are no messages, or no possible current message, the verb is aborted with a warning message to the player.

Once the message number(s) to display has been decided, a loop is entered which lists each message in turn. For each message number given the mailbox is searched for the message with that number. If no message is found with a given number, a warning message is sent to the user.

Printing a message simply involves printing the message number, and the message text - the second item in the mail record.

`none` **@answer** Verb
`none` **@answer** *msg* Verb
`none` **@reply** Verb
`none` **@reply** *msg* Verb
This command is used to reply to a message sent by another player. If no argument is given, the current message number (if valid) is taken to be the message to reply to. Once a valid message has been located in the player's `messages` property, the verb

```
$mail_editor:invoke(2, verb, x[2], {@this.mail_options, @args});
```

is invoked to allow the player to compose a reply, where *x* is the message being replied to.

`num` **receive_message** (`list` *message*) Verb
This function is called by other mail system verbs to place a message in the player's mailbox property, `messages`. If the calling verb has permission to modify the properties on the player, the message is simply added to the end of the current list of messages, and given a message number one greater than the maximum current message number.

A notification is sent to the player, informing him/her that new mail has arrived. The verb returns '`1`' for a successful delivery, and an error code (normally `E_PERM`) otherwise.

`none` **fix_current_message** () Verb
This verb is used to validate the *current message* number property on the player. For example, if a player deletes the current message, then the number stored in the `current_message` property no longer points to a valid message. This verb checks

through all the messages in the mailbox, `player.messages`, until it finds message number `current_message`, or it passes the place in the list where the current message is supposed to be. If this happens, the current message property is set to the next message along in the list. If this does not produce a valid message number, then the current message is taken to be the last message on the list, or '`0`' if there are no more messages.

`none` **@next** () Verb

This command is used to advance the current message pointer to the next message in the player's mailbox. This is done by incrementing the pointer, and then calling *$player:fix_current_message* to validate it. If there is no message after the current message, the pointer is moved off the list.

Once a new current message has been found, it is printed out using the `$player:@read` verb.

`none` **@prev** () Verb

This command is used to backtrack the current message pointer to the previous message in the player's mailbox. This is done by looking for the current message in the message list, and then decrementing the index that expression returns by one place.

Once a new current message has been found, it is printed out using the `$player:@read` verb.

`none` **@renumber** () Verb

This verb is used to renumber the messages in a player's mailbox to all consecutive numbers, starting from one. After checking that the person invoking the verb has permission to perform the renumbering, and there are messages to renumber, the verb executes the following code:

```
newmsgs = {};
i = 0;
for x in (this.messages)
  newmsgs = {@newmsgs, {i = i + 1, x[2]}};
endfor
this.messages = newmsgs;
```

which is fairly self explanatory. The current message number is preserved by getting it's index in the list of messages. This is the number that the message will have after the renumbering operation.

1.2.2 The Mail Room or Mail Editor

When a player wishes to send a mail message, s/he is transported to the mail room to compose the message. This room is a child of the generic editor, and provides all the functions documented in the section on that class. In addition, it provides verbs and properties related specifically to the sending of mail messages. The following section describes each of the `$mail_room` verbs and properties in detail.

`string` **working_on** (`num` *who*) Verb

This verb returns a string describing the mail message currently being worked on by the player identified by *who*. It executes the following code:

```
return this:ok(who = args[1]) && "a letter to " +
               this:recipient_names(who) +
               ((subject = this.subjects[who]) &&
               " entitled \"" + subject + "\"");
```

`list` **parse_invoke** (*args*) Verb

This verb parses the arguments given to `$mail_room:invoke` to determine the type of mail editor invocation required - either produced by an ‘`@answer`’ or a ‘`@send`’ command. In the latter case, the arguments passed to `$mail_room:invoke` are

`invoke(`*rcpts*`, `*verb*`, `*flags* `[, `*subject*`])`

The argument *rcpts* is a list of recipients for the message. This list is parsed using the verb `$mail_room:parse_recipients()`. The *flags* parameter is a list of possible mail options. In this case, it is searched for the `replyto` option, and a variable set if it is found. The parameter *subject* can optionally give the subject for the message. For an ‘`@send`’ command, the returned list from this verb contains the following information:

`{list `*rcpts*`, str `*subject*`, str `*replyto*`, {}}`

For an `@answer` command, the arguments are

`invoke(2, `*verb*`, `*msg*`, `*flags*`)`

For answering a message, a different set of information is given and returned. The verb `$mail_editor:parse_msg_headers` is used to parse the options given by the player into a variable *hdrs*.

Several options can be specified in this way to the '`@answer`' command. as shown below.

`incl`	- include the original message text.
`all`	- reply to everyone in the mail header.
`sender`	- reply to the sender only.
`noincl`	- do not include the original message in your reply.

If the '`incl`' option is given, the original message text is placed into a variable *incl* for returned to the calling verb. Every line is prepended with the characters `>`, as one would expect.

The list shown below is returned to the calling verb:

`{`*hdrs*`,` *include*`}`

where *hdrs* is the list returned by parsing *msg* supplied to the verb, and *include* contains any included message text.

If any problem is encountered in parsing the message headers, or in recognising which command originated the call to this verb, '`0`' is returned.

`none` **init_session** () Verb

`none` **pri*nt** () Verb

`none` **print_msg** () Verb

`none` **message_with_headers** () Verb

`none` **subj*ect:** () Verb

`none` **set_subject** () Verb

`none` **to*:** () Verb

`none` **also*-to:** `cc*:` () Verb

`none` **parse_recipients** () Verb

`none` **recipient_names** () Verb

`none` **make_message** () Verb

`none` **name_list** () Verb

`none` **parse_msg_headers** () Verb

`none` **check_answer_flags** () Verb

`none` **reply-to*:** `replyto*:` () Verb

`none` **send** () Verb

`none` **who** () Verb

`none` **showlists** () Verb

`none` **send_message** () Verb

`none` **subsc*ribe** () Verb

`none` **unsubsc*ribe** () Verb

The following properties are also defined on the `$mail_editor` or `$mail_room` class:

replytos () Property

recipients () Property

subjects () Property

`num` **accept** (`obj` *thing*) Verb

This verb overrides the `$room` class definition. In this case, *thing* is allowed to enter the room only if it is in the list `$mail_room:writers`. In practice, this stops anyone from moving into the room, unless they have used a mail system command that enters their object number into the list of *message writers*. This is necessary because the action of entering the room causes certain mail related state to be setup.

Note that the normal way to enter the mail room is to use the '`@send`', for example. When a player enters the room using one of the mail commands, s/he is added to the list of writers. If, for some reason, the player has to '`@move`' out of the room, then s/he can move back in using '`@move`' because s/he is in the `$mail_room:writers` list.

`none` **enterfunc** (`obj` *player*) Verb

This function is triggered when a player moves into the room by the LambdaMOO server. It performs a number of actions related to setting up the state of the mail room ready for the user to interact with the mail system. The code for this verb is shown below:

```
who = args[1];
recips = this:recipient_names(who);
if (this:message_so_far(who))
  player:tell("You are in the middle of writing a letter to ",
          recips, ".  To discard that message, type 'abort'.");
  this:print_msg(who);
else
  player:tell("Sending a message to ", recips, " ...");
  pass(@args);
endif
```

First, we get the list of recipients for the mail message (specified before the player has entered the room). If the player already has part of a message stored in the mail room, then s/he is allowed to continue the message.

Finally, `pass()` is used to invoke the `$room` class verb `:enterfunc`.

`none` **add_writer** (`obj` *player*, `list` *recipients*) Verb

This verb is used to add a plyer to the list of people composing letters in the mail room. Only players are allowed to do this. If the player is not already composing a message, then s/he is simply added to the list of *writers* for the room. This causes *player* to be added to the `$mail_room:writers` property.

In addition to this, the `recipients` list is added to the list, `$mail_room:recipients`. The original location of the player (before being moved to the mail room) is stored in the `$mail_room:places` list. The subject line and message text for this message is set to the empty string, in the property lists `$mail_room:subjects` and `$mail_room:messages`. The message is also marked as being abortable, by adding a '`0`' to the `$mail_room:abortable` list. This is done because there is no warning given for deletion of an empty message.

If the player is already in the list of people composing messages in the mail room, then the `listset()` primitive is used to change the recipients, subject and abortable property for the message associated with the player. This means that a player can only be composing one message at a time.

Once the above has been performed, the player is moved from his/her current location to the mail room. A suitable message is sent to all the objects in the player's current location telling them what has happened.

Throughout all user interaction in the mail room, the position of the player object reference in the `$mail_room:writers` list is used to index into all the other property lists used when dealing with a mail message.

`list` **recipients** (`obj player`) Verb

This verb returns the list of recipients for the message being composed by *player*.

`list` **message_so_far** (`obj` *player*) Verb

This verb returns the text of the message currently being composed by *player*. This is returned as a list of strings.

`none` **say** (`string` *text*) Verb

This verb overrides the `$room` class definition. It is intercepted to allow the mail program to capture input from the user to append to the message currently being composed. The `$mail_room:say` verb itself just passes the player object reference and *text* to the `$mail_room:add_line` verb for processing.

`none` **emote** (`string` *text*) Verb

This verb overrides the `$room` class definition. It is intercepted to allow the mail program to capture input from the user to append to the message currently being composed. The `$mail_room:say` verb itself just passes the player object reference and *text*, with the player name prepended, to the `$mail_room:add_line` verb for processing.

This allows both '`say`' and '`emote`' input to be captured in the current message.

`none` **add_line** (`obj` *player*, `string` *text*) Verb

This verb is used to add *text* to the current message being composed by *player*. Once a line has been added to a text message, it is not allowed to be immediately aborted. This is set using the `$mail_room:clear_abortable` verb.

`none` **print** () Verb

This verb is a player command used to print the text stored in the current message. The code itself first allows the message to be aborted, using `$mail_room:clear_abortable`, and then calls `$mail_room:print_msg` to print the message text.

`none` **print_msg** (`obj` *player*) Verb

This verb prints the text of the current message being composed by `player`. Again, the code is fairly simple, and is shown below:

```
who = args[1];
who:tell("Your message so far:");
who:tell();
who:tell_lines(this:message_with_headers(who));
who:tell();
who:tell("<End of message>");
```

Note the use of the `$root_class:tell_lines` verb to print the message contents. `$mail_room:message_with_headers` returns a list of strings comprising the mail header, and message text.

`list` **recipient_names** (`obj` *player*) Verb

This verb returns a list of the names of the recipients of the current message being composed by `player`. It retrieves the correct list of recipients from the `$mail_room:recipients` property list, and then maps this from a list of object references to a list of object names using `$mail_room:name_list`.

`none` **pause** () Verb

This is a player command used to temporarily stop composition of a message by a player. A suitable message is printed by the player, and then the `$mail_room:send_back` verb is invoked to send the player back to the room s/he was in when letter composition was started.

`none` **send_back** (`obj` *player*) Verb

This verb is used to return a player to the room s/he was in when a letter composition command was invoked. When a player enters the mail room, the original location of the player is stored in the property list `$mail_room:places`. This verb causes that location to be looked up, and the player is returned to his/her original location. A suitable message is sent to the objects in the player's original location to inform them that s/he has returned from the mail room.

`none` **look_self** () Verb

This verb overrides the `$room` class definition. This is done to inhibit the room name and contents list being printed when the player uses the '`look`' command. Instead, this verb uses `$mail_room:description` as help text, which is sent to the player using `$root_class:tell_lines`.

`none` **help** () Verb

This verb is a player command invoked to get some help on the mail room facilities. This results in the `$mail_room:look_self` verb being called to print the help text stored in the mail room description. Note that this overrides, also, the LambdaMOO help system.

`none` **abort** () Verb

This is a player command used to abort composition of a message, and throw the text away. A check is made to see if the message is abortable. If it is, then the player is sent back to his/her original location, and the message is destroyed using `$mail_room:destroy`.

If this message is marked as being non-abortable, the player is given an '`Are you sure?!`' message. The `$mail_room:abortable` property is used to provide a safety net for players. Typing `abort` once will not throw away the message, unless it is empty.

`num` **abortable** (`obj` *player*) Verb

This verb is used to return the `abortable` status of the message currently being entered by `player`. The `$mail_room:abortable` property list contains a boolean value for each player currently composing a message. The mechanism is included to stop a player accidentally deleting a message by using the '`abort`' command. If the `abortable` flag for a message is set, then the message may be deleted by using the '`abort`' command. If the flag is clear, then deletion is prevented.

One invocation of '`abort`' results in an '`Are you sure?!`' message being printed, and the `$mail_room:abortable` flag for the player's message is reset to allow aborts. A subsequent '`abort`' by the player will then destroy the message.

An empty message can be aborted without any warning. Once a line of text has been entered into a message, this saftey feature is enabled by clearing the `abortable` flag for the message.

`none` **set_abortable** `obj` *player* Verb

This verb is used to set the `$mail_room:abortable` flag for the message currently being composed by *player*. If the flag is set, the message is allowed to be aborted.

`none` **clear_abortable** () Verb

This verb is used to clear the `$mail_room:abortable` flag for the message currently being composed by *player*. If the flag is reset, the message cannoted be aborted.

`none` **exitfunc** `obj` (*player*) Verb

This verb is invoked when a player leaves the mail room. It clears the `abortable` flag for the *player*'s message, and then invokes the `exitfunc` of the parent class.

`none` **destroy** (`obj` *player*) Verb

This verb is used to delete the message currently being composed by *player*. This involves removing the message state from the various property lists stored in the room. The code for this verb is shown below, as the best explanation:

```
who = args[1];
pos = who in this.writers;
this.writers = listdelete(this.writers, pos);
this.recipients = listdelete(this.recipients, pos);
this.messages = listdelete(this.messages, pos);
this.places = listdelete(this.places, pos);
this.abortable = listdelete(this.abortable, pos);
this.subjects = listdelete(this.subjects, pos);
```

`none` **send** () Verb

This verb is a player command invoked to send the current message to the list of recipients. It retrieves the message as a list of strings, and then sends it to each recipient in turn. Note that no third part mail agent is involved. When the '`send`' command is invoked by a player, it delivers mail to the recipients using the verb `$player:receive_message`.

After a message has been sent, it is destroyed. The invoking player is returned to the room s/he was in when the mail system was started.

`list` **message_with_headers** (`obj` *player*) Verb

This verb returns a list of strings, giving the message plus any headers which are added to the front by the mail system. The `$mail_room:make_message` verb is used to produce the list of strings.

`string` **the_subject** (`obj` *player*) Verb

This verb returns the subject text for the message being composed by *player*. This is done by retrieving the correct entry from the property list `$mail_room:subjects`.

`none` **sub*ject** () Verb

This is a player command used to set the subject text of the current message. A simple call is made to the `$mail_room:set_subject` verb, with the arguments supplied by the player in `argstr`.

`none` **to** () Verb

This is a player command used to set the list of recipients for the current message. Without any arguments, the command prints the current list of recipients. If one or more arguments are supplied, then the recipients list for the message is cleared, and the verb `$mail_room:also-to` is invoked to take tge player arguments, and set the message recipients list accordingly.

`none` **set_subject** (`obj` *player*) Verb

This verb is used to set the subject text for the message being composed by *player*.

`none` **also*-to** (`string` *args*) Verb

This verb goes through the list of names given in *args* adding each name to the recipients list for the current message. The code for the verb is shown below:

```
pos = player in this.writers;
recips = this.recipients[pos];
for x in (args)
  r = $string_utils:match(x, players(), "aliases");
  if (r == $failed_match)
    player:tell("There is no player named '", x, "'.");
    return;
  elseif (r == $ambiguous_match)
    player:tell("I don't know which '", x, "' you mean.");
    return;
  endif
  recips = setadd(recips, r);
endfor
this.recipients = listset(this.recipients, recips, pos);
player:tell("Your message is now to ", this:recipient_names(player),
".");
```

Each name in the argument list is matched to a player name. If there is no match, the name is rejected from the list of recipients, and further processing is aborted. If there is a match, the player object reference is added to the list of recipients. If there is an ambiguous match, the name is ignored, and further processing is aborted.

Valid names are added one by one to the recipient list for the message.

`list` **make_message** (`obj` *player*, `list` *to*, `string` *subj*, `list` *msg*) Verb

This verb returns the message as it will be sent to the recipients in the list *to*. It takes the arguments given, and puts together a list of strings that gives the current message being written by *player*. In addition to the supplied arguments, the date and sender's name are added to the message as header.

`string` **name_list** (`list` *names*) Verb

This verb returns an english style list of the names in *names*.

`none` **send_message** (`obj` *from*, `list` *to*, `string` *subj*, `list` *msg*) Verb

This verb is used to actually send a mail message to the players given in the *to* list of recipients. If you wish to send mail from another verb, this is the verb to use. The arguments given are passed to `$mail_room:make_message` for formatting, and then sent to each of the recipients in turn.

The following properties are defined for the `$mail_room` class. Each property contains a list of items, one per player currently composing a letter. The index of the player in the `writers` list is used to locate the correct entries for that player's message in all the other property lists.

subjects Property

This is a list of strings. Each string contains the subject line for a message currently being composed.

abortable Property

This is a list of boolean (number) values. If set, the relevant message can be aborted by the player using the '`abort`' command in the mail room. If not set, the message cannot be aborted.

places Property

This is a list of room object references. When a player enters the mail room, using one of the mail composition commands, his/her current location is stored here. When the player has finished in the mail room, s/he is returned to the original room.

recipients Property

This is a list of lists of object references, referring to players. Each entry in the list looks like the following :

```
{player1, player2, ..., playerN}
```

The message currently being composed will be sent to *player1* thru *playerN* when it is '`sent`'.

messages Property

This property stores the actual message text being composed by the player. Each entry in the list is a list of strings, one per line of the message.

writers Property

This is the list of players currently engaged in composing a message in the mail room. Every time a player enters the mail room using one of the mail commands, his/her object reference is added to this list. If a player enters the mail room and is already on this list, this indicates a resumption of a previously '`pause`'d mail session.

1.2.3 The generic Editor Class

This class provides a basic MOO code editing facility for those players who do not have access to an external editor such as Emacs. It provides basic facilities for editing and compiling MOO code on verbs, but cannot be considered complete or extensive.

1.2.4 The System Object Class

The *system object* class is the fundamental cornerstone of the whole LambdaMOO database. It is the first item in the database, and hence has an object number of '`#0`'. It has no verbs defined on it, but a number of interesting properties that are used to determine the behaviour of parts of the LambdaMOO server and Core functions.

In particular, properties defined on the root class can be referred to using the '`$`' notation as a shorthand. For example, if I define a property called

```
#0.foobar = #1234
```

then I can refer to '#1234' thereafter using the shorthand

```
$foobar
```

Shorthand references are provided for all of the base classes. If a new public class is added, it should be given a shorthand by adding a suitable property to the system object, '#0'. In this way, players can refer to the shorthand when creating objects that belong to the new class.

For example, if `foobar` were the name of a new class, then I could use

```
@create $foobar named gadget
```

to create an instance of the class `foobar`.

Shorthand notation is provided in the following properties. Most of them refer to public classes. A few refer to other, global values that are used in the database.

editor Property

mail_room Property

object_utils Property

lock_utils Property

list_utils Property

command_utils Property

player Property

wiz Property

prog Property

code_utils Property

help Property

nothing Property

This should be used similarly to a `void` pointer in C.

failed_match Property

This is the value used in all matching verbs to indicate that nothing matched.

ambiguous_match Property

This value is used in matching verbs to indicate that more than one possible result was found.

perm_utils Property

building_utils Property

string_utils Property

news Property

note Property

letter Property

container Property

thing Property

exit Property

room Property

root_class Property

This points to the class from which everything else is descended.

The other properties defined for the system object are listed below, with details given as to their function and derivation.

wizard_mail_recipients Property

This property is a list of object numbers. It gives a list of people who will receive mail sent by players to the wizard. This can loosely be interpreted as being a list of players that are involved in running the game, and are not strictly wizards.

gripe_recipients Property

This property is a list of object numbers. It gives a list of people who will receive mail sent by players using '`@gripe`'. This can loosely be interpreted as being a list of players that are involved in running the game, and are not strictly wizards.

dump_interval Property

This property specified the interval between database dumps, in seconds. The default value is 600 seconds, which is every ten minutes. A database dump should be initiated at regular intervals to provide a backup in case the LambdaMOO server crashes, or the host machine goes down.

welcome_message Property

This property is a list of strings containing the message sent to players when they log in to the LambdaMOO server. It should be used to convey simple help information to first time players, as well as making urgent announcements that you wish everyone to see.

player_start Property

This is an object reference of the starting room for new players. When a player is created, using the '`create`' command when initially connecting to the LambdaMOO server, s/he is placed in the `$player_start` room. This is also set as the default home for a new player. If you wish to place new players into a different starting room, then you should change the object reference stored in `$player_start`.

player_class Property

This is an object reference for the class that newly created players belong to by default. The initial value of this property is `$player`. If you create an enhanced player class that you wish all new players to belong to, you should change the value of this property to be the object reference of the new, enhanced player class.

1.2.5 The Container Class

A container is a an object that can hold other object inside itself. This is similar to the idea of a room, except that a container has no notion of exits, and can only hold things; it cannot hold players.

The following verbs are defined for the container class.

`none` **d*rop** (`obj` *object*) Verb
`none` **in*sert** (`obj` *object*) Verb
`none` **p*ut** (`obj` *object*) Verb

This verb puts *object* into the container. The container is first checked to verify that it is open, if not an error message is printed.

For example, if you have a container named *pipe* and an object named *tobacco*, you could enter:

```
put tobacco in pipe
```

If *pipe* is open, then *tobacco* will be put into the *pipe*. If you look at *pipe* you should see:

```
pipe
Contents:
  tobacco
```

none **g*et** (obj *object*) Verb
none **ta*ke** (obj *object*) Verb
none **re*move** (obj *object*) Verb
This does the opposite of the put/insert/drop commands. To remove the *tobacco* from the *pipe* you would enter:

```
remove tobacco from pipe
```

When you look at *pipe* now you should see:

```
pipe
It is empty.
```

none **open** () Verb
This verb will open the container and allow objects to be put into it (via the *put* verb). This verb sets the property *opened* to '1'.

none **@lock_for_open** (obj *object*) Verb
This verb will lock the container with *object*. The container can only be opened if the player is holding *object*, or the *object* is the player trying to open the container. The container will remained locked until it is unlocked (see below).

none **@unlock_for_open** () Verb
This verb will remove the lock set by *@unlock_for_open*. It can only be run by the owner of the container.

num **is_openable_by** (obj PLAYER) Verb
This verb uses the `$lock_utils:eval_key` verb to determine if the container can be opened by the player. The verb will return '1' if the player has permission to open the container, and '0' otherwise.

`none` **close** () Verb

This is the opposite of *open*. If the pipe is already close, an error message is printed.

`none` **tell_contents** () Verb

This verb displays the contents of the container. If the container is empty, the message '`It is empty`' is displayed.

`none` **set_opened** (`int` *number*) Verb

This verb is called by *open* and *close*, with the arguments '`1`' and '`0`' respectively. The property *opened* is set to either '`1`' (opened) or '`0`' (closed).

`none` **@opacity** (`int` *number*) Verb

Opacity determines when/if you can look at the contents of the container. There are three levels of opacity:

```
0 - transparent
1 - opaque
2 - black hole
```

When the opacity is set to 0, you can see the contents when the pipe is open or closed. When the opacity is set to 1, you can only see the contents when the container is open. If opacity is set to 2, you can never see the contents when looking at the container.

The syntax for '`@opacity`' is:

```
@opacity container is #

where '#' is either 0, 1 or 2.
```

`int` **set_opaque** (`int` *number*) Verb

This verb is called by '`@opacity`'. It sets the property *opaque* to be either 0, 1 or 2.

1.2.6 The Note Class

The note class provides a handy way of communicating in MOO, whether you want to leave somebody a quick message, or perhaps leave a clue for solving an adventure. The note class can also provide a good starting base for more specialized objects. The letter and newspaper objects are examples of this.

`none` **r*ead** () Verb

This verb first checks to see if the note can be read by the person who is attempting to read the it. If they have permission to read it, the note text, which is stored in the property `text`, is shown to the player.

`none` **er*ase** () Verb

This verb erases all text stored on the note. Only the owner of a note can erase it.

`none` **wr*ite** (`string` *message*) Verb

Appends the string *message* to the note text. This function simply adds another line to the list stored in the note property 'text'. A carriage return is inserted between each list entry, therefore the command:

```
write "Zaphod was here." on note
```

would be read as

```
Zaphod was here.
```

whereas the commands

```
write "Zaphod" on note
write "was" on note
write "here." on note
```

would be read as

```
Zaphod
was
here.
```

`none` **del*ete** (`string` *line-number*) Verb

`none` **rem*ove** (`string` *line-number*) Verb

Deletes the given line number from the note. Only the owner of the note may modify it. Negative numbers can be given to delete lines counting from the end of the note, hence '`delete -5 from note`' would delete the 5th line from the bottom.

`list` **text** () Verb
Returns a list of strings containing the message text on the note. Called by the verb `:r*ead`.

`num` **is_readable_by** (`obj PLAYER`) Verb
This verb uses the `$lock_utils:eval_key` verb to evaluate the key, stored in the `encryption_key` property, and determine if the note is readable or not by the player. This verb will return a '`1`' if it is readable, or a '`0`' if it is not.

`none` **encrypt** (`string ENCRYPTION`) Verb
This verb will encrypt a message so that it can only be read if the encryption constraints are satisfied. Refer to the '`@lock`' command for details on the syntax for encryption. Encryption is stored in the `encryption_key` property. You must have permission to do this.

`none` **decrypt** () Verb
This verb will remove any existing encryption from a note, allowing it to be read by anybody. You must have permission to do this.

The following properties are defined for the `$note` class.

encryption_key Property
This is encryption string for the note. It is formed within the verb `:encrypt` by parsing its input with `$lock_utils:parse_keyexp`. The verb `:is_readable_by` then uses this to determine whether or not the note can be read by calling `$lock_utils:eval_key`.

text Property
This is a list of strings where the body of the note message is kept. Each entry in the list corresponds to one line in the note text.

1.2.7 The Newspaper Class

To be written.

1.2.8 The Letter Class

The letter class is a subclass of the note class. A letter object provides all of the functions of a note, with the addition of a '`burn`' command. This command will recycle the note after it has been read by the recipient, completely removing it from the database. The letter is read, written, erased and encrypted just as if it were simply a note object.

`none` **burn** Verb

This verb first checks to see if the letter is readable by the person who is attempting to burn it. If it is, the letter will be completely destroyed. If it is not readable, the command will fail. The code for this is shown below for illustration:

```
if (this:is_readable_by(player))
  player:tell(this.name,
         " might be damp, in any case it won't burn.");
else
  player:tell(this.name,
      " burns with a smokeless flame, and leaves no ash.");
  player.location:announce(player.name,
      " stares at ", this.name, " and it catches alight.");
  recycle(this);
endif
```

1.2.9 The Generic Programmer Class

1.2.10 The Help Database Class

The LambdaCore database provides an extensive help facility. As well as command syntax, it provides other useful information on a variety of other MUD topics. The `$help` class is used to store help text and verbs used to implement the help system. A verb on the player class, `$player:help` is used to give the player access to the help database.

Each entry in the help database is stored as a property on the `$help` class. By doing this, an index can be generated by using the `properties()` primitive to list all the properties on `$help`. No other properties are stored on the `$help` class.

The '`help`' command, defined on the `$player` class, is described below.

`none` **help** Verb
`none` **help** *topic* Verb
`none` **help index** Verb

This verb, part of the player class, is used to gain access to the help system. The first form is used to obtain help on the topic `summary`, which is a top level help with pointers to other topics in the database.

If the third form, '`help index`' is used, the verb `$help:print_index` is invoked to print a list of all the help topics currently available.

The most complex case is the second form, where help is being requested on a particular topic. The relevant code segment is shown below:

```
topics = $help:find_topics(dobjstr);
if (topics == {})
  player:tell("Sorry, but no help is available on \"",
        dobjstr, "\".");
  return;
elseif (length(topics) > 1)
  player:tell("Sorry, but the topic-name \"",
            dobjstr, "\" is ambiguous.",
   " I don't know which of the following topics you mean:");
  for x in (topics)
    player:tell("   ", x);
  endfor
  return;
else
  topic = topics[1];
  if (topic != dobjstr)
    player:tell("Showing help on \"", topic, "\" ...");
    player:tell();
  endif
endif
$help:print_topic(topic);
```

First, the verb `$help:find_topics` is called. This takes the *topic* given by the player, and searches through the help database index for possible matches. This returns a list of possible topics in the variable *topics*.

If the list is empty, than no help was available for the topic given by the player, and the verb ends.

If the list contains more than one item, the player is told what matched, and asked to be more specific next time.

If just one help topic was found, it is printed out, using the verb `$help:print_topic`.

The `$help` class has three verbs defined on it which are used to implement the help system. They are described in detail below:

`list` **find_topics** (`string` *handle*) Verb

This verb is used to search the list of topics on the `$help` class for one that matches *handle*. The list of topics is given by the list of properties defined on the `$help` class. In the simple case, if the *handle* matches one property name exactly, then this name is returned.

The player is allowed to omit the leading '`@`' on commands. This is checked for in this verb by adding an '`@`' temporarily, and then checking for this in the topic list, along with the un-prefixed *handle*.

If no direct match is found, the verb looks at all the properties stored on the `$help` class, and tries to make a match using `index()` between the *handle*, the *handle* with an '`@`' prepended and all the property names. Every possible match is returned by this verb, so that the '`help`' command can give the list to the user.

If no matches are found, an empty list is returned.

`none` **print_index** () Verb

This verb is used to print an alphabetical list of help topics, as given by the properties stored on the `$help` class. The code for this function is shown below, as it illustrates one method of sorting strings in MOO, using lists.

```
topics = setremove(properties(this), "summary");
buckets = "abcdefghijklmnopqrstuvwxyz";
else_bucket = length(buckets) + 1;
counts = names = {};
for i in [1..else_bucket]
  names = {@names, {""}};
  counts = {@counts, 1};
endfor
keys = names;
count = 1;
for name in (topics)
```

```
    if (name && index(".@", name[1]))
      key = name[2..length(name)];
    else
      key = name;
    endif
    k = index(buckets, key[1]) || else_bucket;
    bucket = keys[k];
    count = counts[k];
    i = 1;
    while (i <= count && key > bucket[i])
      i = i + 1;
    endwhile
    names = listset(names, listappend(names[k], name, i - 1), k);
    keys = listset(keys, listappend(keys[k], key, i - 1), k);
    counts = listset(counts, count + 1, k);
  endfor
  sorted = {};
  for i in [1..length(names)]
    sorted = {@sorted, @names[i][2..counts[i]]};
  endfor
  sorted = {@sorted, "", "", ""};
  n = length(sorted) / 4;
  su = $string_utils;
  for i in [1..n]
    player:tell(su:left(sorted[i], 20),
                su:left(sorted[i + n], 20),
                su:left(sorted[i + n + n], 20),
                sorted[i + 3 * n]);
  endfor
```

none **print_topic** (string *topic*) Verb

1.3 The LambdaCore Utility Classes

The following utility classes are provided in the LambdaCore database:

`String Utils. ($string_utils)`
  This class provides a number of verbs containing string utility functions.

`Building Utils. ($building_utils)`
  This class provides a couple of verbs used in building commands.

`Permissions Utils.  ($perm_utils)`
: This class provides verbs used in manipulating permissions on objects.

`Code Utils.  ($code_utils)`
: This class provides some useful general purpose verbs.

`Command Utils.  ($command_utils)`
: This class provides some verbs used for command line parsing.

`List Utilities.  ($list_utils)`
: This class provides a couple of verbs used in manipulating lists.

`Matching Utils.  ($match_utils)`
: This class provides verbs used in matching names to objects.

`Trignometric Utilities ($trig_utils)`
: This class provides some trigonometric functions and other mathematical utilities.

`Lock Utilities.  ($lock_utils)`
: This class provides verbs used in dealing with locks on objects.

`Object Utilities.  ($object_utils)`
: This class provides some useful verbs used in manipulating objects.

`Time Utilities ($time_utils)`
: This provide some time manipulation utilities.

`Gender Utilities ($gender_utils)`
: These provide some gender based utilities.

1.3.1 The String Utilities Class

The string utilities class, `$string_utils` defines a number of verbs useful for performing operations on strings.

`list` **explode** (`list` *subject* [, `str` *break*]) Verb

This verb is used to explode a string into a list of substrings separated by runs of *break* as the delimiting character. If *break* is not specified it defaults to space. As an example:

```
$string_utils:explode("This    is a   test");
```

will return the list

```
{"This", "is", "a", "test"}
```

`str` **from_list** (`list` *list* [, `str` *separator*]) Verb

This verb is used to implode the string representations of the elements of a list into a single string, each pair being separated by *separator*. which defaults to the empty string. This function is essentially the inverse of `:explode`. Note that the elements of *list* need not be strings themselves, as the function `tostr` is applied to each element before it is catenated. As an example, to reassemble the list above into the original string, with just one space between words, you could write:

```
oldstr = $string_utils:from_list({"This", "is", "a", "test"}, " ");
```

`object` **match** (`str` *string* [, `list` *obj-list*, `str` *prop-name*]`*`) Verb

This verb is used to search for a match to *string* in all the specified properties of all the specified objects, returning either object, `$ambiguous_match` or `$failed_match`.

Each *obj-list* should be a list of objects or a single object, which is treated as if it were a list of that object. Each *prop-name* should be a string naming a property on every object in the corresponding *obj-list*. The value of that property in each case should be either a string or a list of strings.

The argument string is matched against all of the strings in the property values.

If it exactly matches exactly one of them, the object containing that property is returned. If it exactly matches more than one of them, `$ambiguous_match` is returned.

If there are no exact matches, then partial matches are considered, ones in which the given string is a prefix of some property string. Again, if exactly one match is found, the object with that property is returned, and if there is more than one match,

`$ambiguous_match` is returned.

Finally, if there are no exact or partial matches, then `$failed_match` is returned.

`str` **from_value** (`item` *value* [, `str` *quote_strings* [, `num` *list_depth*]]) Verb
This verb is used to print the given value into a string. Note that *quote_strings* defaults to 0 (false), and `list_depth` defaults to 1. This is best described by example:

```
$string_utils:from_value(v) gives:

value result 5 "5" 5 + 3 "8" "5 + 3" "5 + 3" {} "{}" {1,2,3} "{1,
2, 3}" {1, {2, 3}, 4} "{1, {list}, 4}" {"1", {2, 3}, 4}
"{1, {list}, 4}"

$string_utils:from_value(value, 1) gives:

value                       result
5                      "5"
5 + 3                  "8"
"5 + 3"                "\"5 + 3\""
{}                     "{}"
{1,2,3}                "{1, 2, 3}"
{1, {2, 3}, 4}         "{1, {list}, 4}"
{"1", {2, 3}, 4}       "{\"1\", {list}, 4}"


$string_utils:from_value(value, 0, 2) gives:

value                       result
5                      "5"
5 + 3                  "8"
"5 + 3"                "5 + 3"
{}                     "{}"
{1,2,3}                "{1, 2, 3}"
{1, {2, 3}, 4}         "{1, {2, 3}, 4}"
{"1", {2, 3}, 4}       "{1, {2, 3}, 4}"

$string_utils:from_value( value, 1, 2 ) gives:

value                       result
5                      "5"
5 + 3                  "8"
"5 + 3"                "\"5 + 3\""
{}                     "{}"
{1,2,3}                "{1, 2, 3}"
{1, {2, 3}, 4}         "{1, {2, 3}, 4}"
{"1", {2, 3}, 4}       "{\"1\", {2, 3}, 4}"
```

It is interesting to note that `:from_value` calls itself recursively, to evaluate lists.

`str` **pronoun_sub** (`str` *text* [, `obj` *who*]) Verb

This verb is used to substitute the pronoun properties of *who* in all occurances of %s,%o,%p,%r in *text*. *who* is optional, and defaults to player. Also '`%n`', '`%d`', '`%i`', '`%t`', '`%%`' are substituted by `player`, `dobj`, `iobj`, `this` and `%` respectively. Further, '`%(propname)`' is substituted by *who*`.propname`. Capitalised versions of each of these are: '`%S`', '`%O`', '`%P`', '`%R`', '`%N`','`%D`', '`%I`', '`%T`' and '`%(Propname)`'. The full list is given below:

Code	Property	Pronoun	Defaults
%%			%
%s	*who*.ps	subjective	he, she, it
%S	*who*.psc	subjective	He, She, It
%o	*who*.po	objective	him, her, it
%O	*who*.poc	objective	Him, Her, It
%p	*who*.pp	possessive	his, her, its
%P	*who*.ppc	possessive	His, Her, Its
%r	*who*.pr	reflexive	himself, herself, itself
%R	*who*.prc	reflexive	Himself, Herself, Itself
%n	*who*.name		
%N	*who*.name		(capitalised)
%d	dobj.name		
%D	dobj.name		(capitalised)
%i	iobj.name		
%I	iobj.name		(capitalised)
%t	this.name		
%T	this.name		(capitalised)
%(xyz)	*who*.xyz		
%(Xyz)	*who*.xyz		(capitalised)

`str` **space** (`num` *num* [, `str` *str*]) Verb

This verb returns a string of *num* occurences of *str*. *str* is optional, and defaults to space (" ").

```
:space(8)         returns   "        "
:space(8, "z")    returns   "zzzzzzzz"
:space(4, "<>")   returns   "<><><><>"
```

`str` **right** (`str` *text*, `num` *len* [, `str` *fill*]) Verb

This verb is used to right justify *text* in a string of length *len*. *fill* is the optional fill character, which defaults to space. This function calls `$string_utils:space` (described above). It is interesting to note that text need not be a string, as `tostr()` is applied to it before justification.

```
:right("help", 9)      returns   "     help"
:right("me", 9, ".")   returns   ".......me"
:right(200, 9, "*")    returns   "******200"
:right({1,2}, 9)       returns   "   {list}"
```

`str` **centre** (`str` *text*, `num` *len* [, `str` *fill*]) Verb

This verb (aka `$string_utils:center`) is used to centre *text* in a string of length *len*. *fill* is the optional fill character, which defaults to space. This function calls `$string_utils:space`, (described above).

```
:centre("help", 9)      returns   "  help   "
:centre("me", 9, ".")   returns   "...me...."
:centre(200, 9, "*")    returns   "***200***"
:centre({1,2}, 9)       returns   " {list}  "
```

`str` **left** (`str` *text*, `num` *len* [, `str` *fill*]) Verb

This verb is used to left justify *text* in a string of length *len*. *fill* is the optional fill character, which defaults to space. This function calls `$string_utils:space`, (described above). It is interesting to note that text need not be a string, as `tostr()` is applied to it before justification.

```
:left("help", 9)      returns   "help     "
:left("me", 9, ".")   returns   "me......."
:left(200, 9, "*")    returns   "200******"
:left({1,2}, 9)       returns   "{list}   "
```

`str` **english_list** (`list` *list* [`str` *empty* `str` *and* `str` *sep* `str` *penum*]) Verb

This verb is used to return a list of things as an english sentence. Note that `tostr()` is applied, so things in the list need not necessarily be strings.

The optional arguments allow you to control how the list is presented. *empty* is the string returned if the list is empty. The default is '`nothing`'. *and* is the string to use instead of '` and `' in the list. A common usage of this is replace '` and `' with '` or `'. *sep* is the separator to use between words, the default being '` ,`'. Finally, *penum* is the string to use after the penultimate elements, before the '`and`'. The default is to have a comma without a space.

```
:english_list({})                         =  "nothing"
:english_list({"cat"})                    =  "cat"
:english_list({"cat","dog"})              =  "cat and dog"
:english_list({"cat","dog","pig"})        =  "cat, dog, and pig"
:english_list({1})                        =  "1"
:english_list({{1, 2, 3}, "Hi", 300})     =  "{list}, Hi, and 300"
```

`str` **from_seconds** (`num` *secs*) Verb

This verb is used to return a string showing time in units of days, hours, minutes or seconds.

```
:from_seconds(0)        returns  "0 seconds"
:from_seconds(2)        returns  "2 seconds"
:from_seconds(61)       returns  "a minute"
:from_seconds(130)      returns  "2 minutes"
:from_seconds(7000)     returns  "an hour"
:from_seconds(1000000)  returns  "11 days"
```

`num` **find_prefix** (`str` *subject*, `list` *choices*) Verb

This verb is used to find a string from a list of *choices* which has the prefix *subject*.

For example, if choices = `{"hat", "hand", "face", "help"}` then...

```
:find_prefix("f",  choices)  ==  3
:find_prefix("g",  choices)  ==  0
:find_prefix("h",  choices)  ==  $ambiguous_match
:find_prefix("he", choices)  ==  4
:find_prefix("ha", choices)  ==  $ambiguous_match
```

`str` **trim** (`str` *text* [, `str` *what*]) Verb

This verb is used to trim leading and trailing characters from the string *text*. *what* is the optional character to trim, and defaults to space.

```
:trim("help")             returns  "help"
:trim("   help   ")       returns  "help"
:trim(">> help <<", ">")  returns  " help <<"
```

`str` **capitalise** (`str` *text*) Verb
`str` **capitalize** (`str` *text*) Verb
`str` **cap_fast** (`str` *text*) Verb

This verb is used to capitalise its argument. Note that it depends on `strcmp()` returning the difference of the first non-matching letters!

```
:cap_fast("hello")     returns    "Hello"
:cap_fast("Ezeke")     returns    "Ezeke"
```

`str` **strip_chars** (`str` *subject*, `str` *stripped*) Verb

This verb is used to remove the characters in the string *stripped* from the string *subject*.

```
:strip_chars("The quick brown fox", " ")      =  "Thequickbrownfox"
:strip_chars("The quick brown fox", "aeiou")  =  "Th qck brwn fx"
```

`list` **match_player** (`str` *name*, ..., [`obj` *meobj*) Verb

`obj` **match_player** (`str` *name*, `obj` *meobj*) Verb

This verb is used to match one or more *name*s to player objects in the database. Note that *name* need not be complete, nor in the same case as the player's actual name. If the *name* given is '`me`', then the value of `player`, or *meobj* is returned. If *meobj* is not a player, then `$failed_match` is returned.

In the case where a name is not matched against a player name or alias, the value `$failed_match` is returned. If a name matches more than one player, then the value `$ambiguous_match` is returned.

For example:

```
Verb Call                                 Returned
---------                                 --------
:match_player("blip")                     #35
:match_player("bli")                      #35
:match_player("blip", "Fred")             {#35, $failed_match}
:match_player("blip", "Ezeke")            {#35, #99}
:match_player("me", #35)                  #35
:match_player("me", #1234)                $failed_match
:match_player("b")                        $ambiguous_match
```

This assumes that players named '`blip`' and '`Ezeke`' exist in the database, but '`Fred`' doesn't. Additionally, there is more than one player whose name starts with '`b`'.

`list` **words** (`str` *text*) Verb

This verb is used to split a piece of *text* into words, in the same wat that the command line parser turns `args` into `argstr`. This verb differs from using, for example,

`$string_utils:explode` in that it recognises a quoted piece of text as being one word. For example:

```
Verb Call                                Returned
---------                                --------
:words ("There is a dog.")               {"There", "is", "a", "dog."}
:words ("A \"red nose\"")                {"A", "red nose"}
:words ("")                              {}
:words ("Hello")                         {"Hello"}
```

`str` **cap_property** (`obj` *what*, `str` *prop*, [, `num` *ucase*])) Verb

This verb returns *what.prop*, but capitalised if either *ucase* is true, or the property name specified is capitalised. If *prop* is blank, the verb returns the value of *what*`:title()`. If *prop* is not specified on *what* or is otherwise irretrievable, then an error is returned.

If capitalisation is indicated by *ucase* being true, we return *what*`.`*prop*`c` if that exists, otherwise *what.prop* is capitalised in the normal fashion using `$string_utils:capitalise`. A special case exists if we are trying to retrieve the name of a player. This is never capitalised.

`num` **index_delimited** (`str` *string*, `str` *target* [, `num` *casem*]) Verb

This verb works in the same way as the primitive `index()`, except that it only matches on occurrences of *target* in *string* that are delimited by word boundaries. That is, it will not match on occurrences that are preceded or followed by an alphanumeric.

For example,

```
Verb Call                                        Returned
---------                                        --------
:index_delimited ("Hi there", "he")                 0
:index_delimited ("Hi there", "hi")                 1
:index_delimited ("Hi there", "hi", 1)              0
:index_delimited ("Hi there", "Hi", 1)              1
```

`num` **to_value** (`str` *expr*) Verb

This verb is used to convert a string into a value. It is the opposite operation to `$string_utils:from_value`. As with that verb, this is best explained by examples:

```
$string_utils:to_value(s) gives:
```

s	result
"5"	5
"5 + 3"	"5 + 3"
{}	$\boxed{\text{error}}$
{1,2,3}	$\boxed{\text{error}}$
"{1,2,3}"	{1,2,3}

`list` **columnize** `columnise` (`list` *items*, `num` *n* [, `num` *width*]) Verb

This verb is used to turn a one-column list of items into an *n* column list. *width* is the last character position that may be occupied; it defaults to a standard screen width of 79 characters.

For example, to tell a player a list of number in three columns, use the following code:

```
player:tell_lines($string:utils:columnise ({1, 2, 3, 4, 5, 6, 7},
3));
```

`list` **parse_command** (`str` *command*, *who*) Verb

This verb is used to parse a command line, in the same way that the built in command line parser does the job. It returns an horrendous list of stuff, that I'm not quite sure about yet.

`list` **match_str*ing** (`str` *input*, `str` *match*, . . .) Verb

This verb performs wildcard matching of '`match`' to *input*, using the '`*`' character as a wildcard. It returns a list of what the `*`s actually matched. Note that this verb will not catch every match, if there are several different ways to parse the input. More than one wildcard string may be given. If a numeric argument is given, it is taken to indicate whether case sensitivity should be used.

For example:

```
;$string_utils:match_string(\"Jack waves to Jill\",\"* waves to *\")
⊣{\"Jack\", \"Jill\"}
```

`str` **uppercase** `lowercase` (`str` *string*) Verb

This verb returns either the uppercase or lowercase version of *string*, as appropriate.

No properties are defined on the `$string_utils` class.

1.3.2 The Permissions Utils Class

The permission utilities class, `$perm_utils` provides a couple of useful verbs for dealing with permissions of objects, i.e.. the ability of a player to read or access another object.

`num` **controls** (`obj` *who*, `obj` *what*) Verb

Returns true, '`1`', if *who* is the owner of *what*, or if *who* is a *wizard*, else returns false, '`0`'.

`str` **apply** (`str` *perms*, `str` *mods*) Verb

This verb is used to modify the permissions given in *perms* with the modifiers shown in *mods*. The following modifiers are possible:

`!`	- negate a permission.
`+`	- add a permission.
`-`	- remove a permission.

Consider the following examples:

```
Verb Call                          Returns
---------                          -------
;$perm_utils:apply("rw", "!w")     "r"
;$perm_utils:apply("rw", "!r")     "w"
;$perm_utils:apply("rw", "-w")     "r"
;$perm_utils:apply("rw", "")       ""
;$perm_utils:apply("rw", "x")      "x"
;$perm_utils:apply("rw", "+x")     "rwx"
```

No properties are defined on the `$perm_utils` class.

1.3.3 The Code Utils Class

The code utils class is somewhere to hang code functions. These are defined as verbs of the `$code_utils` class.

`list` **parse_propref** (`str` *string*) Verb

This function parses *string* as a MOO-code property reference, returning

{*object-string*, *prop-name-string*}

for a successful parse and false otherwise. It always returns the right object-string to pass to, for example, `$room:match_object()`.

The following examples show how this verb works:

```
:parse_propref("me")               returns   0
:parse_propref("me.name")          returns   {"me", "name"}
:parse_propref("$room.location")   returns   {"#3", "location"}
:parse_propref("$thing")           returns   {"#0", "thing"}
:parse_propref("$player:tell")     returns   0
```

`list` **parse_verbref** (`str` *string*) Verb

This function parses *string* as a MOO-code verb reference, returning

{*object-string*, *verb-name-string*}

for a successful parse and false otherwise. It always returns the right object-string to pass to, for example, `$room:match_object()`.

The following examples show how this verb works:

```
:parse_verbref("me")              returns   0
:parse_verbref("me:title")        returns   {"me", "title"}
:parse_verbref("$room:blow_up")   returns   {"#3", "blow_up"}
:parse_verbref("$thing")          returns   0
:parse_verbref("player:tell")     returns   {"player", "tell"}
```

`none` **show_object** (`obj` *obj*) Verb
`none` **show_property** (`obj` *obj*, `str` *pname*) Verb
`none` **show_verbdef** (`obj` *obj*, `str` *vname*) Verb

These verbs are the ones used by the '`@show`' command to display details of an object, a property on an object, or a verb on an object. Below is some sample code used to display details on the paranoid property of ezeke:

```
fullpropref = "ezeke.paranoid";
here = player.location;
prop = $code_utils:parse_propref(fullpropref);
if (!prop)
  player:tell("Could not parse string.");
else
  obj = here:match(prop[1]);
  if (obj < #0)
    player:tell("Could not match '", prop[1], "'.");
  else
    $code_utils:show_property(obj, prop[2]);
  endif
endif
```

`none` **find_verbs_containing** (`str` *pattern*, `num` *start_obj*, `num` *count*) Verb

This verb prints to the player the name and owner of every verb in the database whose code contains *pattern* as a substring. It starts at the object number *start_obj* (as a number, not an object) and goes to the end of the database. *count* specifies how many verbs have been found so far. Because it searches the entire database, this function may not be finished when it returns; it may have forked a task to continue the job.

For example:

```
>;$code_utils:find_verbs_containing ("show_property", 0, 0)
⊣#4:@s*how, line 18 [Wizard (#2)]
```

1.3.4 The Command Utilities Class

The command utilities class, `$command_utils` provides a couple of useful verbs for dealing with player commands, as follows:

`num` **object_match_failed** (`obj` *result*, `str` *string*) Verb

This function tells the player of a failed match, and returns true if so. If *result* is valid it returns false, without telling the player anything. For example, (where `#118` has been recycled):

```
Result            String  Ret  Player sees...
------            ------  ---  --------------
$failed_match     "blip"   1   I see no "blip" here.
$ambiguous_match  "blip"   1   I don't know which "blip" you mean.
```

```
#118              ""       1   #118 does not exist.
#28               "blip"   0
```

This could be used in the following way, in the implementation of a 'bop' command, defined as a verb on a player:

```
/* bop <any> */
here = player.location;
victim = here:match(dobjstr);
if (!$command_utils:object_match_failed(victim, dobjstr))
  player:tell("You bop ", dobj.name, " over the head!");
  dobj:tell(player.name, " bops you over the head!");
  here:announce_all_but({player,dobj},
  player.name + " bops " + dobj.name + " over the head!");
endif
```

none **player_match_result** (`list` *results*, `list` *strings*, [`str` *prefix*]) Verb

This verb is used to inform the player of the results of trying to match the player names in *string* with players in the database. It assumes that *results* are the result of a `$string_utils:match_player(strings)` and prints out nasty messages to the player concerning *strings* that didn't match with any player in teh database. It returns a list, in the following format:

{*our_result*, *player1*, ..., *playerN*}

where *our_result* is an overall result, set to '`true`' if some string didn't match, followed by the object reference numbers of players that matched. An optional third argument, *prefix*, gives an identifying string to prefix to each of the nasty messages.

For example, supposing there are only players named '`blip`' and '`blipster`' in the database:

```
Results            Strings     Ret      Player Sees....
-------            -------     ---      ---------------
{$nothing}         {"George"}  0        nothing
{$failed_match}    {"George"}  1        "George" is not the
                                            name of any player.
{$ambiguous_match} {"blip"}    1        "blip" could refer to any
                                            of a number of people."
{#28}              {"blip"}    {0, #28} nothing
```

The following code fragment from `$player:page` shows how you might use this function in your own programs:

```
who = $string_utils:match_player(args[1]);
if ($command_utils:player_match_result(who, args[1])[1])
  return;
```

The variable *args[1]* contains the name of the player we are trying to page

No properties are defined on the `$command_utils` class.

1.3.5 The List Utilities Class

The list utilities class, `$list_utils` defines a number of functions that manipulate lists.

`list` **map_prop** (`list` *objs*, `string` *prop*) Verb

This function returns a list of the values of a specified property *prop* on each of the objects in the list *objs*. If *prop* does not exist on one of the objects, an error occurs.

As an example, the following code will list all the locations that have at least one player in them. It uses the `:map_prop` verb to generate a list corresponding to each player's location. Set addition is used to provide a list of the location names without duplication. Note that the `players()` primitive returns a list of all the player objects in the database.

```
locs = $list_utils:map_prop(players(), "location");
rooms = {};
for x in (locs)
  rooms = setadd(rooms, x.name);
endfor
player:tell("Inhabited locations are ",
            $string_utils:english_list(rooms), ".");
```

`list` **map_verb** (`list` *objs*, `string` *verb* [, `list` *varg*]) Verb

This function returns a list of the return values of a specified verb *verb* on each of the objects in the list *objs*. *varg* is optional and defines the arguments to *verb*; it defaults to the empty list `{}`. If *verb* does not exist on one of the objects, an error occurs.

As an example, the following code will list all the connected players, using the `:title` verb on each to do so. It uses the `:map_verb` function to generate a list of each of the player's *title strings*.

```
cp = $list_utils:map_verb(connected_players(), "title");
player:tell("Connected players are ",
              $string_utils:english_list(cp), ".");
```

`list` **assoc** (*target*, `list` *list*, [`num` *index*]) Verb

This verb requires *list* to be a list of lists. It returns the first element of *list* whose own *index*th element is *target*. If not supplied, *index* defaults to '`1`'.

Consider the following examples:

```
Verb Call                         Returned
---------                         --------
:assoc(1, {{2,3}, {1, 2}})        {1,2}
:assoc(1, {{2,3}, {2, 1}})        {}
:assoc(1, {})                     {}
```

Consider a more interesting example. Suppose that you have a list of heights for a number of players, in the following format:

`{{`*name1*`,` *height1*`}, ..., {`*nameN*`,` *heightN*`}}`

and you want to search through the list to find out how tall a particular player is. This can be done using the following code, assuming that the variable *heights* is the list of '`{`*name*`,` *height*`}`' pairs, and *ourplayer* is the player whose height you wish to look up:

```
player:tell(ourplayer, "is ",
    $list_utils:assoc (ourplayer, heights)[2], "ft high");
```

Suppose that *heights* is defined as follows:

```
{{"blip", 2}, {"Ezeke", 6}, {"Geezer", 10}}
```

For different lookups, we would get the following results:

```
ourplayer                    Results
blip                         {"blip", 2}
Ezeke                        {"Ezeke", 6}
```

```
Fred                        {}
```

`num` **iassoc** (*target*, `list` *list*, [`num` *index*]) Verb

This verb works similarly to `$string_utils:assoc` except that it returns an index into *list*, rather than the actual matching element from *list*. If no matching element is found, it returns '`0`'. Consider the following examples:

```
Verb Call                       Returned
---------                       --------
:assoc(1, {{2,3}, {1, 2}})      2
:assoc(1, {{2,3}, {2, 1}})      0
:assoc(1, {})                   0
```

`list` **sort_alist** (`list` *list*, [`num` *index*]) Verb

This verb requires *list* to be a list of lists. It sorts the *list* into ascending order, using the *index*th element of each list element as the sort key. For example:

```
Verb Call                         Returned
---------                         --------
sort_alist({{5}, {2}, {3}})       {{2}, {3}, {5}}
sort_alist({})                    {}
sort_alist({{2,3},{5,1},{3,6}},2) {{5,1},{2,3},{3,6}}
```

`list` **map_pronoun_sub** (`list` *objects*, *string*) Verb

This verb returns the result of calling `$string_utils:pronoun_sub` for each object in the list *objects* on the *string* supplied. For example, where '`#28`' is named '`blip`', and '`#35`' is named Ezeke:

```
our_objs={#28,#35};
our_msg="Hello %n";
our_output=$list_utils:map_pronoun_sub(our_objs, our_msg);
player:tell_lines (our_output);
```

Executing this short program would produce the following output

```
>run_test_program
⊣Hello blip.
⊣Hello Ezeke.
```

`list` **slice** (`list` *list*, [`num` *index*]) Verb

This verb returns a list of the *index*th elements of the elements of *list*. If *index* is not given, it defaults to '`1`'. For example,

```
Verb Call                           Returns
---------                           -------
:slice ({{1,2}, {3,4}, {5,6}}})     {1,3,5}
:slice ({{1,2}, {3,4}, {5,6}}},2)   {2,4,6}
:slice ({}}                         {}
```

No properties are defined on the `$list_utils` class.

1.3.6 The Building Utils Class

This class provides a couple of utility functions used in building rooms and exits in the core database structure. The verbs provided are used mainly by the building command '`@dig`', although they can, of course, be used by other verbs.

`none` **set_names** (`obj` *object*, `string` *spec*) Verb

The function is used to set both the name of an object and any aliases that the programmer needs. It expects 2 arguments: first, the object reference whose name is to be set, and then a string which contains the name and any aliases as a comma delimited string.

`none` **make_exit** (`string` *spec*, `obj` *source*, `obj` *dest*) Verb

This is a function that is used to create a named exit with given source and destination rooms. The code will do a protection check to verify that an exit can be created and which if any links specified can be made.

The following will try to create an exit called '`Out`' owned by the calling player and then link that exit to #25 as the source (exit from) and #26 as the destination (entrance to).

```
$building_utils:make_exit("Out", #25, #26);
```

1.3.7 The Lock Utilities Class

This class provides some useful functions related to the use of *locks* for objects in the database. The LambdaCore database supports a simple but powerful notation for specifying locks on objects, encryption on notes, and other applications. The idea is to describe a constraint that must be satisfied concerning what some object must be or contain in order to use some other object.

The constraint is given in the form of a logical expression, made up of object numbers connected with the operators '`and`', '`or`', and '`not`' (written '`&&`', '`||`', and '`!`', for compatibility with the MOO programming language).

These logical expressions (called *key expressions*) are always evaluated in the context of some particular *candidate* object, to see if that object meets the constraint. To do so, we consider the candidate object, along with every object it contains (and the ones those objects contain, and so on), to be '`true`' and all other objects to be '`false`'.

As an example, suppose the player blip wanted to lock the exit leading to his home so that only he and the holder of his magic wand could use it. Further, suppose that blip was object #999 and the wand was #1001. blip would use the '`@`lock' command to lock the exit with the following key expression:

```
me || magic wand
```

and the system would understand this to mean

```
#999 || #1001
```

That is, players could only use the exit if they were (or were carrying) either #999 or #1001.

There is one other kind of clause that can appear in a key expression:

`?` *object*

This is evaluated by testing whether the given object is unlocked for the candidate object; if so, this clause is true, and otherwise, it is false. This allows you to have several locks all sharing some single other one; when the other one is changed, all of the locks change their behavior simultaneously.

The internal representation of key expressions is stored in the property `.key` on every object.

1.3.7.1 Key Representation For Locks

Objects are represented by their object numbers and all other kinds of key expressions are represented by lists. These lists have as their first element a string drawn from the following set:

```
"&&"      "||"      "!"      "?"
```

For the first two of these, the list should be three elements long; the second and third elements are the representations of the key expressions on the left- and right-hand sides of the appropriate operator. In the third case, '`!`', the list should be two elements long; the second element is again a representation of the operand. Finally, in the '`?`' case, the list is also two elements long but the second element must be an object number.

As an example, the key expression

```
#45  &&  ?#46  &&  (#47  ||  !#48)
```

would be represented as follows:

```
{"&&", {"&&", #45, {"?", #46}}, {"||", #47, {"!", #48}}}
```

1.3.8 Verbs for Manipulating Locks and Keys

The following utility verbs, defined on the `$lock_utils` class can be used by the programmer to manipulate locks on objects and key expressions.

num **eval_key** (`list` *key*, `obj` *who*) Verb

This verb evaluates the key expression *key*, in the context of the candidate object *who*. It returns '`1`' if the *key* will allow *who* to pass, otherwise it returns '`0`'. The simple examples below are intended to show the ways in which this verb can be invoked:

```
Verb Call                      Returned
---------                      --------
:eval_key(#35, #35)               1
:eval_key({"!", #35}, #35)        0
:eval_key({"!", #35}, #123)       1
```

The *key* expression is given as a list, in the format described above.

`list` **parse_keyexp** (`string`*keyexp*, `obj` *who*) Verb

This verb parses the string *keyexp* and returns the internal representation of the expression, for use with `$lock_utils:eval_key`. It is used, for example, by the verb `$player:@lock` to parse the text given by a player into a key expression to store on the `.key` property of an object.

`none` **init_scanner** () Verb
`none` **scan_token** () Verb
`none` **match_object** () Verb
`none` **canonicalize_spaces** () Verb
`none` **parse_E** () Verb
`none` **parse_A** () Verb

These verbs are used by the verb `$lock_utils:parse_keyexp` to perform it's parsing functions. They can be called from user programs, but are really part of the `:parse_keyexp` verb.

`string` **unparse_key** (`list` *keyexp*) Verb

This verb takes the key expression *keyexp* and returns the input expression that could be used to obtain that key. It is the reverse operation to `$lock_utils:parse_keyexp`.

A few properties are defined on the `$lock_utils` class, but these are only used internally by the lock evaluation routines, and are of no real interest to the programmer.

1.3.9 The Object Utilities Class

This class provides some utilities used to ascertain information about objects.

`num` **contains** (`obj` *container*, *target*) Verb

This verb returns '1' if *target* is in any way contained by *container*. This includes being directly in the contents list of the object, and also in the contents list of any object that is contained by *container*. This verb is a partner to the `:match` verb, as they both embody the same concept.

`list` **all_properties** (`obj` *object*) Verb

This verb returns a list of names of all the properties of *object*. This includes properties defined on the parent(s) of this object. Recall that properties defined on the parent(s) of an object do not appear in the set given by the `properties(`*thing*`)` primitive in MOO. The `:all_properties` verb is useful for seeing the properties that an object actually has values for.

`list` **all_verbs** (`obj` *object*) Verb

Returns a list of all the names of the verbs defined for *object*. This includes verbs defined on the parent(s) of this object. Again, as for the `:all_properties` verb, `:list_all_verbs` provides the complete list of verbs that define an object's behaviour. Using the `verbs(`*thing*`)` primitive in MOO lists only those verbs actually *defined* on *thing*.

`list` **ancestors** (`obj` *object*) Verb

This verb returns a list of the parent objects of this object. This is all the objects that lie above this object in the hierarchy. All objects have the `$root_class` in their list of parents. For example, the `$newspaper` class object has the `$note` class, the `$thing` class and the `$root_class` as it's parents.

This verb can be used by an object to tell if another object is descended from the same ancestor class. Consider, for example, a `generic zapper` object. This is an object which can be used to `shoot` at other people, with amusing and sometimes unexpected results. Bearing in mind that it is unfair to try and `zap` someone who is unarmed, the verb `zapper:shoot` looks at the possessions of the victim, and decides whether s/he is armed. If this is the case, then the victim is fair game and deserves all s/he gets. If not, the zapper refuses to shoot, with embarrassing results for the aggressive attacker.

Considering the implementation of the `zapper:shoot` verb, we could determine if the victim has a zapper by comparing the `parent`s of the assailant's zapper and each of the victim's possessions. However, this would only work if the zappers were first generation children of the same parent - the `generic zapper` class.

Suppose, however, that the crafty player has built a `super zapper` class, with radar sights, based on the `generic zapper` class. In this case, the test fails because the children of the `super zapper` class are not first generation descendants of the original `generic zapper` class.

A better way to code this test is to look at the `:ancestors` of the objects, and see if the `generic zapper` class is numbered amongst them. If this is the case, then we can definitely know who is armed and who isn't.

`list` **descendants** (`obj` *object*) Verb

Returns a list of all this object's descendants, going down the object hierarchy.

`obj` **has_verb** (`obj` *object* `string` *vname*) Verb

This verb returns the object number of *object* if it defines the verb *vname*. If *object* does not define *vname*, then the object number of any ancestor of *object* that does define the verb is returned. If the verb is not defined on *object* or any of it's ancestors, then '`0`' is returned.

`obj` **has_property** (`obj` *object* `string` *pname*) Verb

This vernb returns '`1`' if *object* has a property *pname*, else it returns '`0`'.

1.3.10 The Matching Utilities

The matching utilities class, `$match_utils` provides several useful verbs for matching strings to objects. The following facilities are provided:

`obj` **match** (`str` *string*, `list` *where*) Verb

This is a general purpose matching verb. It tries to match *string* against the list of objects in *where*, if *string* does not match any of the special cases, listed below.

`me`	This matches with the value of the `player` variable, i.e. the player who is running the verb.
`here`	This returns the value of the `player.location` property.
`""`	If a `null` argument is supplied, this verb returns the value of the `caller` variable.
`$`*class*	This returns the value of the corresponding property on the system object, `#0`. This is used to match with, for example, `$player`, `$exit` etc.
`#`*num*	This returns the value of *num* as an object reference, as long as the object referred to is valid, and meets one of the following conditions: • The object is in the object that called this verb.

- The object is in the same location as the player.
- The object is being carried by the player.
- The object is in the same location as the caller.
- The object is owned by the player.

string In the general case, a match is attempted using the `$string_utils:match` verb, using the value of *where* as the list of objects to match against. The `aliases` property is used to match the string against.

If no match is found using the above rules, a more general algorithm is used. This recognises forms such as the following:

person`'s` *object*
: This matches against the item *object* in the inventory of *person*. The special case where *person* is specified by using '`my`' is catered for. Note that `$match_utils:match` is used to match the *person* string.

number object
: This form allows you to specify a particular object in situations where the *object* string would match more than one item. The string *number* is something like '`first`', '`second`', ... or '`1st`', '`2nd`', ...

These two forms can be combined, as show in the examples below. Here we assume that wizard(#2) is carrying two newspapers, with object numbers `#10` and `#23`. Wizard is currently standing in a room with object reference `#11`. The verb calls are made by the Wizard, so that referring to '`me`' or '`my`' refers to the wizard.

```
Verb Call                                  Returns
---------                                  -------
:match ("me", {})                             #2
:match ("here", {})                           #11
:match ("", {})                               $nothing
:match ("$player", {})                        #4
:match ("wizard", {})                         $failed_match
:match ("wizard", {#2})                       #2
:match ("my newspaper", {})                   #10
:match ("wizard's newspaper", {})             $failed_match
:match ("wizard's newspaper", {#2})           $ambiguous_match
:match ("wizard's first newspaper", {#2})     #10
```

```
:match ("wizard's second newspaper", {#2})     #23
:match ("first newspaper", {#2})               $failed_match
:match ("first newspaper", {#10, #23})         #10
:match ("second newspaper", {#10, #23})        #23
```

`obj` **match_nth** (`str` *string*, `list` *where*, `num` *n*) Verb

This verb is used to match the *n*th occurrence of *string* in the list of objects given in *where*. This verb is used, for example, by `$match_utils:match` to pick out one particular object from a list with more than one possible match.

This verb looks at the `aliases` property of each of the objects in the *where* list, and returns the *n*th match. If there is no *n*th match, then `$failed_match` is returned. The idea behind this verb is to allow you to specify particular objects in situations where a normal match would return *$ambiguous_match*. This verb uses `index()` to do it's matching, and so returns objects which partially match *string*.

`num` **match_verb** (`str` *verb*, `obj` *what*, *args*) Verb

This verb returns '`1`' if *verb* is found on the object *what*. The code for this is shown below:

```
vrb = args[1];
what = args[2];
ret = what:(vrb)(args[3]);
return ret != E_VERBNF;
```

Note that the verb being matched is executed with the *args* given, but the result returned by the verb is lost. If the verb is not found, then '`0`' is returned.

`list` **match_list** (`str` *string*, `list` *where*) Verb

This verb returns a list of objects taken from the list *where* that match with *string*. The `aliases` property of each object is checked to see if it can be matched against *string*. If it can, then it is added to the returned list. This verb is used in cases where you wish to know all the objects that match a particular *string*. This verb uses `index()` to do its matching, and so returns objects which partially match *string*.

1.3.11 The Trigonometric Utilities

This class provides various numerical routines, including trigonometric routines, generation of number sequences, and permutations and combinations.

The trigonometric routines all take a parameter *x* in degrees. Result returned are to four decimal places, multiplied by 10000, so that they can be represented by numbers. For example:

```
Verb Call                        Result
---------                        ------
:sin(0)                              0
:sin(90)                         10000
:sin(45)                          7063
:sin(270)                       -10000
```

The following verbs are available:

`num` **sin** (`num` *x*) Verb

This verb returns the sine of *x*, where *x* is in degrees. The returned value is to four decimal places, multiplied by 10000.

`num` **cos** (`num` *x*) Verb

This verb returns the cosine of *x*, where *x* is in degrees. The returned value is to four decimal places, multiplied by 10000.

`num` **tan** (`num` *x*) Verb

This verb returns the tangent of *x*, where *x* is in degrees. The returned value is to four decimal places, multiplied by 10000.

`num` **xsin** (`num` *x*) Verb

`num` **xcos** (`num` *x*) Verb

These two verbs are used by the above verbs to compute the sin, cos and tan values using Taylor's Theorem. They are not really of use to the average programmer.

`num` **factorial** (`num` *x*) Verb

This verb returns the factorial, using the traditional recursive method. Note that *n* must be greater than or equal to 0.

`num` **pow** (`num` *x*, `num` *n*) Verb

This verb returns *x* raised to the *n*th power. Note that *n* must be greater than or equal to 0. Again, a recursive method is used to generate the result.

`list` **exp** (`num` *x* [, `num` *n*]) Verb

This verb calculates an *n*th order Taylor approximation for `e^x`. *n* defaults to 5. The result returned is in the format:

{*integer part*, *fractional part*}

For example,

```
>;$trig_utils:exp(1)
 ⊣{2, 71666}

>;$trig_utils:exp(2, 6)
 ⊣{7, 35555}
```

`list` **fibonacci** (`num` *n*) Verb

This verbn calculates the Fibonacci sequence to the *n*th term, and returns the result as a list of numbers. Note that *n* must be greater than or equal to zero.

For example:

```
>;$trig_utils:fibonacci(10)
 ⊣ {0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55}
```

`num` **geometric** (`num` *x*, `num` *n*) Verb

This verb calculates the value of the geometric series at *x* to the *n*th term. i.e., it approximates `1/(1-x)` when `|x| < 1`. This, of course, is impossible in MOO, but someone may find it useful in some way. *n* defaults to 5. Note that *n* must be greater than or equal to zero.

`list` **div** (`num` *n*, `num` *q*) Verb

This verb returns a decomposition of *n* by *q* using the division algorithm. The returned list contains the divisor and the remainder.

For example:

```
Verb Call                     Returned
---------                     --------
;$trig_utils:div(5,1)          {5, 0}
;$trig_utils:div(5,2)          {2, 1}
```

```
;$trig_utils:div(5,3)         {1, 2}
;$trig_utils:div(5,4)         {1, 1}
;$trig_utils:div(5,5)         {1, 0}
```

`num` **combinations** (`num` *n*, `num` *r*) Verb

This verb returns the number of ways one can choose *r* objects from *n* distinct choices. For example:

```
Verb Call                           Returned
---------                           --------
;$trig_utils:combinations(5,1)         5
;$trig_utils:combinations(5,2)         10
;$trig_utils:combinations(3,2)         3
```

`num` **permutations** (`num` *n*, `num` *r*) Verb

This verb returns the number of ways possible for *r* objects to be ordered given *n* distinct locations,using the formula

```
P(n,r) =  n!
         ----
         (n-r)!
```

For example:

```
Verb Call                           Returned
---------                           --------
;$trig_utils:permutations(5,1)         5
;$trig_utils:permutations(5,2)         20
;$trig_utils:permutations(3,2)         6
```

`list` **simpson** (`list {num` *a*`, num` *b*`}, list {num` *f*(*a*)`, num` *f*((*a*+*b*)/2)`,`
`num` *f*(*b*)`})` Verb

Given two endpoints, *a* and *b*, and the function's value at *a*, *(a+b)/2*, and *b*, this verb will calculate a numerical approximation of the integral using Simpson's rule. The answer is returned in the form

{*integer part*, *fractional part*}

`list` **abs** (`num` *x*) Verb

This verb returns the absoluate value of *x*. Note that this is available in the server as the `abs()` primitive, so this verb is redundant.

`list` **parts** (`num` *n*, `num` *q* [, `num` *i*]) Verb

This verb returns a decomposition of *n* by *q* into integer and floating point parts. *i* gives the number of digits after the decimal point. If not given, *i* defaults to 5. For example:

```
Verb Call                       Returned
---------                       --------
;$trig_utils:parts(5,1)          {5, 0}
;$trig_utils:parts(5,2)          {2, 50000}
;$trig_utils:parts(3,2,2)        {1,50}
```

1.3.12 The Time Utilities

The time utilities class, `$time_utils` provides a few useful functions for dealing with dates and times. The following verbs are provided:

`str` **day** (*time*) Verb

This verb takes a time specification in *time*, and returns the full name of the day. This verb expects *time* to be the result of a either the `time()` primitive or *ctime()*. In the first case, *time* is a number. In the second case, *time* is a string.

For example,

```
>;$time_utils:day(time())
 ⊣ "Friday"
```

The names for the days are stored on a property list called `days` on the `$time_utils` class. This is indexed from a list of abbreviated day names, stored in a property list called `dayabbrs`.

`str` **month** (*time*) Verb

This verb takes a time specification in *time*, and returns the full name of the month. This verb expects *time* to be the result of a either the `time()` primitive or *ctime()*. In the first case, *time* is a number. In the second case, *time* is a string.

For example,

```
>;$time_utils:month(time())
 ⊣"July"
```

The names for the months are stored on a property list called `months` on the `$time_utils` class. This is indexed from a list of abbreviated month names, stored in a property list called `monthabbrs`.

`string` **ampm** (*time* [`,num` *precision*) Verb

This verb takes a time specification in *time*, and returns string containing a time in the form

```
[h]h[:mm[:ss]] {a.m.|p.m.}
```

time is either the result of the `time()` or *ctime()* primitives, and is either a number or a string respectively. If given, *precision* specifies the accuracy required:

`1`	Only the hours are returned.
`2`	The hours and minutes are returned.
`3`	The hours, minutes and seconds are returned.

If not given, then *precision* defaults to minutes. For example,

```
Verb Call                   Returns
---------                   -------
:ampm(time())                "8:58 p.m."
:ampm(time(), 1)             "8 p.m."
:ampm(time(), 2)             "8:58 p.m."
:ampm(time(), 3)             "8:58:23 p.m."
```

`num` **to_seconds** (`str` *string*) Verb

This verb takes a *string* in the form

```
hh:mm:ss
```

for example, created using the expression

```
$string_utils:explode(ctime(time))[4]
```

and returns the number of seconds elapsed since 00:00:00.

num **sun** ([num *time*]) Verb

I can't work out what this verb is supposed to do. I include the code for those interested persons who might find it useful.

```
r = 10000;
h = r * r + r / 2;
time = args == {} ? time() | args[1];
t = (time + 120) % 86400 / 240;
s = 5 * ((time - 14957676) % 31556952) / 438291;
phi = s + t + this.corr;
cs = $trig_utils:cos(s);
spss = ($trig_utils:sin(phi) * $trig_utils:sin(s) + h) / r - r;
cpcs = ($trig_utils:cos(phi) * cs + h) / r - r;
return (this.stsd * cs - this.ctcd * cpcs - this.ct * spss + h) /
r - r;
```

num **from_ctime** (str *string*) Verb

This verb, given a string such as that returned by `ctime()`, returns the corresponding time-in-seconds-since-1970 time returned by `time()`, or `E_DIV` if the format is wrong in some essential way. Note that `ctime()` doesn't return a time zone, yet it arbitrarily decides whether it's standard or daylight savings time.

The following properties are defined on the `$time_utils` class:

stsd Property
ctcd Property
ct Property
corr Property

These properties are used by the mysterious `$time_utils:sun()` verb. Becuase of this, I have no comment to make on what information they contain, or how they are used.

dayabbrs Property

This property contains a list of day abbreviations, such as are found in the string returned by `ctime()`. They are used to produce an index into the property `days`, that contains the full names of each day of the week.

days Property

This property contains a list of the seven full names of the days of the week.

months Property

This property contains a list of the twelve full names of the months of the year.

monthabbrs Property

This property contains a list of month name abbreviations, such as are returned from `ctime()`. They are used to produce an index into the property `months`, that contains the full names of each month of the year.

zones Property

This contains a list of some time zones, for some random purpose as yet unknown. If anyone can enlighten me.....

1.3.13 The Gender Utilities

The gender utilities class, `$gender_utils` contains a couple of verbs used for manipulating the gender pronouns of objects.

`num` **set** (`obj` *object*, `str` *gender*) Verb

This verb is used to set the pronoun properties of *object*, according to the *gender* specified. *gender* is a string: one of the strings in the property list `$gender_utils.genders`, the list of rcognized genders.

The verb checks *gender* against the gender list, and sets the pronouns on *object* with strings taken from property lists stored on `$gender_utils`. If the gender change is successful, the (full) name of the gender (e.g., '`female`') is returned. `E_NONE` is returned if gender does not match any recognized gender. Any other error encountered (e.g., `E_PERM`, `E_PROPNF`) is likewise returned and the object's pronoun properties are left unaltered.

`num` **add** (`obj` *object* [`,str` *perms*] [`, obj` *owner*]) Verb

This verb is used to add pronoun properties to *object* if they are not already there. The *owner* and *perms* arguments allow you to optionally specify the permissions and owner of the gender properties that are being created. If not given, *owner* defaults to the object's owner, and *perms* defaults to '`rc`'.

The following properties are defined on the `$gender_utils` class:

pronouns Property
This property holds a list of the pronoun properties that can be given to an object. It is used by the `$gender_utils:add` verb when adding the gender properties to an object.

genders Property
This property holds a list of legal genders for objects, as strings.

ps Property
This property holds the subjective pronouns for each different gender. If a gender is given by `$gender_utils.genders[`*x*`]`, then the corresponding gender pronouns is given by `$gender_utils:ps[`*x*`]`.

po Property
This property holds the objective pronouns for each different gender. If a gender is given by `$gender_utils.genders[`*x*`]`, then the corresponding gender pronouns is given by `$gender_utils:po[`*x*`]`.

pp Property
This property holds the possessive pronouns for each different gender. If a gender is given by `$gender_utils.genders[`*x*`]`, then the corresponding gender pronouns is given by `$gender_utils:pp[`*x*`]`.

pq Property
This property holds the possessive pronouns, in the noun form, for each different gender. If a gender is given by `$gender_utils.genders[`*x*`]`, then the corresponding gender pronouns is given by `$gender_utils:pq[`*x*`]`.

pr Property
This property holds the reflexive pronouns for each different gender. If a gender is given by `$gender_utils.genders[`*x*`]`, then the corresponding gender pronouns is given by `$gender_utils:pr[`*x*`]`.

psc Property
This property holds the capitalised subjective pronouns for each different gender. If

a gender is given by `$gender_utils.genders[`*x*`]`, then the corresponding gender pronouns is given by `$gender_utils:psc[`*x*`]`.

poc Property

This property holds the capitalised objective pronouns for each different gender. If a gender is given by `$gender_utils.genders[`*x*`]`, then the corresponding gender pronouns is given by `$gender_utils:poc[`*x*`]`.

ppc Property

This property holds the capitalised possessive pronouns for each different gender. If a gender is given by `$gender_utils.genders[`*x*`]`, then the corresponding gender pronouns is given by `$gender_utils:ppc[`*x*`]`.

pqc Property

This property holds the capitalised possessive pronouns, in their noun form, for each different gender. If a gender is given by `$gender_utils.genders[`*x*`]`, then the corresponding gender pronouns is given by `$gender_utils:pqc[`*x*`]`.

prc Property

This property holds the capitalised reflexive pronouns for each different gender. If a gender is given by `$gender_utils.genders[`*x*`]`, then the corresponding gender pronouns is given by `$gender_utils:prc[`*x*`]`.

2 Common Questions - How to do Useful Things

To be written.

Verb Index

Property Index

Table of Contents