The evolving TecfaMOO Book
- Part II: Technical Manual

Daniel K. Schneider [1] Richard Godard [2] Gustavo Glusman [4] with K.Block [5]
Terrence M. Drozdowski, [3]
Jenifer Tennison [6]

October 30, 1996

[1]TECFA, School of Psychology and Educational Sciences University of Geneva, <Daniel.Schneider@tecfa.unige.ch>
[2]University of Montreal, <rgodard@divsun.unige.ch>
[3]Arizona State University, Phoenix (aka xymox).
[4]Weizmann Institute of Science, Rehovot, Israel, <Gustavo@bioinfo.weizmann.ac.il>
[5]Computer Consultant, Los Angeles
[6]Psychology Department, University of Nottingham

DISCLAIMER: The Current Draft has been written by Daniel Schneider (rather a beginner). Some information may be wrong at this time. Use your own judgement ! (Co-authors are wizards who actually implemented things in our MOO at various moments)

Programmers, **please check out section 9** on programming at **TECFA**`MOO` , quite a few things are different (improved) from LambdaCore.

The **TECFA**`MOO` is a text-based virtual reality[1]. It is a Virtual Space for Educational Technology, Education, Research and Life at TECFA, School of Psychology and Education, University of Geneva, Switzerland.

[1]http://tecfa.unige.ch/edu-comp/WWW-VL/eduVR-page.html

Contents

Part I

HOW TO CONNECT AND OTHER BASICS

Chapter 1

Introduction

This chapter should be consulted in detail by anybody new to the MOO world. Others may have a quick glance at sections concerning just **TECFAMOO** . Conceptual “MOO” issues are addressed in The evolving TecfaMOO Book, Part I[1]

> “MOO, the programming language, is a relatively small and simple object-oriented language designed to be easy to learn for most non-programmers; most complex systems still require some significant programming ability to accomplish, however.” ([Curtis, 1993, 15])

Conventions and style of this manual

General Style: At least the introductory sections are pretty informal and mostly example based.

Variables: words included in “<.....>” denote variables. E.g. if you see something like “<name>” in a command, you have to replace “name” by something else and DON’T type the “<”.....“>” !

Literal Strings: ”....” have to be typed as double quotes. They delimit a string!

Moo commands: ’....’ Single quotes within text paragraphs are used to denote MOO commands. E.g. “You should type ’look here’ ...” means enter “look here” from your keyboard.

Moo objects “<obj>” refers to an object, e.g. #123 or $thing or <name> (something you can see in the room you are or that you carry)

This manual has been basically written for a general public. However it is heavily **TECFAMOO** biased. Sections dealing with **TECFAMOO** only will be labeled as such, like this:

> 1.4.7 [**TECFAMOO**] Getting a user name

[1] http://tecfa.unige.ch/moo/book1/tm.html

1.1 Acknowledgements

A lot of materials have been drawn from various mailing list on various MOOs. We have done or best to acknowledge the original authors (and will hunt down missing references).

Material is "stolen" so far from:

- Thwarted Efforts (E MOO, ...)
- Defender (World MOO, ...)
- EricM (BioMOO, Diversity University, others....)

Please let us know if you want copied material removed, but consider that this text is available for free to the MOO community. Unless otherwise stated all materials have only been subject to formatting, content has been left unaltered.

Also note, that we may in the future rewrite those sections, but this method allows us to get a quick start.

1.2 Very very minimal communication and navigation

First of all you need to able to connect to a MOO (see section 1.3 for instructions).

Here is some basic information on how to do the most basic actions in a MOO. You can always type *'help'* to get an index of help topics. To get specific help about communication, for example, you can type 'help communication'.

MOOs are organized around "rooms", i.e. you are always in some location. So a first thing you need to learn is how to look at the room you're in: Just type *'look'*. To get information on the objects around you, you can type *'look <object>'*.

To communicate with people in the same room you're in, you can "talk" to them by typing something like:

```
say Hello, there.
   - or -
"Hello, there
```

To know who else is connected at the time, type *'@who'*. You get also information on where they are, for how long they have been connected, and when they were active last. To communicate with people not in your room, you can 'page' them like in the following example:

```
page Jane Do you have some free time?
```

If you want to go to another room, you can walk though an "exit" *by typing the exit's name*. Exits are normally listed when you 'look' like in the following example:

On most MOOs, even guests are allowed to "join" a person, i.e. jump to the same location. But please, before you join a person page her and ask if you are welcome !

```
@join jane
```

If you want to look at things, use the "look" command, like in the following examples:

```
>look
Daniel's Office
You see open boxes, a desk and other brand new office furniture
Obvious Exits: Atrium (to The Tecfa Atrium) and Tower (to Tower).

>l colin
Colin
A young man, having a lot of fun inside the MOO.
He stared so much at his screen that he begins to
have square eyes. Work, all that matters is work !
He is awake, but has been staring off into space for 2 hours.
```

Finally, to logoff, type '@quit'

1.3 Connecting to a MOO and Getting a User Name

In order to connect to a MOO you need 3 things:

1. A MOO client (telnet will do for a first trial)
2. A MOO address
3. How to connect with a given client

1.3.1 MOO Addresses

See The TecfaMOO page for details on how to connect to **TECFAMOO** in various ways. Here we will just use this MOO as an example on how to connect to a MOO. All you need to know is the Internet Name of the machine the MOO runs on (e.g. `tecfamoo.unige.ch`) and a port number (e.g. `7777`). Most Moos run on either port 7777 or 8888, but don't rely on this !!.

TECFAMOO is at: tecfamoo.unige.ch 7777[2]
The WWW interface is at the same address: i.e. Open URL: http://tecfamoo.unige.ch:7777[3].

This means that **TECFAMOO** runs on the machine tecfamoo.unige.ch on port 7777. Each MOO client has a different way of entering machine names and port numbers:

1.3.2 Entering a MOO address to the client

1. For a simple **telnet** connection: Just type 'telnet tecfamoo.unige.ch 7777' on Unix or Vax machines, 'c tecfamoo.unige.ch' on most DOS machines. Under Windows (e.g. WTnvt) and MacIntosh telnet programs you have to enter the port number in a different box. Note that by default the port in those boxes is 23 (ordinary telnet), you don't want that ! However, avoid simple telnet *at all costs* if you can, since telnet does not separate input and output!

[2] telnet://tecfamoo.unige.ch:7777
[3] http://tecfamoo.unige.ch:7777

2. On **Muddweller** open 'File' - 'New' first. Then under the 'Configure Menu' enter the host name (or IP number) and the port number. Finally 'Open Connection' in the 'Configure Menu'. Next you may want to 'Save' this configuration in the 'File' Menu. As in MudWin or tkmoo light use the small type-in window at the bottom to enter commands !

3. On **MudWin** simply open 'File' - 'New' and fill in the 2 boxes. Note that you enter commands in the single small line at the top or the window! (or at the bottom if you configured your client this way).

4. On the emacs rmoo-client you have to do 2 things:

   (a) Add first a MOO to the known list of MOOs: either edit the .rmoo_worlds file (I prefer that) or type the 'M-x rmoo-worlds-add-new-moo' command and answer the questions in the mini-buffer.

   (b) Type 'M-x rmoo' (followed by the MOO World name)

   (c) Note M-x on simple terminals is pressing the ESC key, releasing it and then press the x key. On X terminals hold down both the ALT and the X key.

Some clients are allready configured with a few MOO addresses when you unpack them. It should be easy to add the MOOs you prefer. You also may want to configure your client to use a fixed font, since some MOOs use ascii "graphics" a lot.

1.3.3 Logging into a MOO

Before connecting to a MOO you have to be sure that either guest connections work or that you can obtain a "character" (user identity).

Most advanced clients will allow to automatize connection for a given user name. We will leave this to the reader. In the normal case (without specifying a user name and password to the client) you will see something like this when your client opens a connection to a MOO:

```
OOOOOO OOOOO OOOOO OOOOO    OO     M   M    MM     MM    Questions ?
  O    O     O     O       O  O    MM MM   M  M   M  M   schneide@divsun.unige.ch
  O    OOO   O     OOO    OOOOOO   M M M   M  M   M  M   MooMail: to kaspar
  O    O     O     O      O    O   M   M   M  M   M  M
  O    OOOOO OOOOO O      O    O   M   M    MM     MM

  A Virtual Space for EDUCATIONAL TECHNOLOGY, EDUCATION, RESEARCH and LIFE at
  TECFA, School of Psychology and Education, University of Geneva, Switzerland.

  The TecfaMOO Project WWW page (including 'how to use' information):
  http://tecfa.unige.ch/tecfamoo.html

  Connect as a registered person by typing "connect (name) (password)"
  Guests: type 'connect guest' - Type 'who' to see who is connected.
```

To connect simply type "connect" followed by your user name and password. If you don't have a login carefully read the instructions on the page and check if you can request a user name or if guest login is allowed. Some MOOs will allow you to request a character via the "@request" command once you are logged in as a guest, in some MOOs the "request" command before you log in is enabled. Usually instructions are given on the first screen.

```
connect <user name> <password>
         e.g.
connect Anna rumpel
         or
connect guest
```

Some MOOs will ask questions for guest connections, e.g. ask you for a name. Don't ignore those questions. Again here is an example from **TECFAMOO** :

```
<!-- Connected --!>
[Please enter your name:]
testing
```

1.3.4 Getting a user name

Some Moos allow people to get themselves a 'user id' or character without any condition attached. Others review requests befog giving out new user ids.

The standard procedure for requesting a user is:

1. Connect to a MOO as 'guest' (no password required), enter:

   ```
   connect guest
   ```

2. Request a character:

   ```
   @request-character <name> <email-address>

   <name> = a name without blank spaces <email> = a regular email
   address, confirmation and a password will be sent to this address
   ```

   Note that on some MOOs, additional questions may be asked by the registration procedure.

On some MOOs characters can be created from the login screen with the 'create' command, but on most MOOs this is disabled.

1.4 Selection, Installation and Use of a Mud/MOO Client

When considering a client you have to evaluate several questions:

1. On which machine does it run (.e.g Unix, Windows, Mac)

2. Are you using a telnet connection to work from a host machine ?
3. What kind of terminal /screen do you use ? (e.g. ASCII terminal, X Windows, size ...)
4. What do you want to use it for ? E.g. just for communication & navigation, building or even programming.

The clients we discuss here are not necessarily the best in all areas, but they are widely used and work on standard networking setups. All have been used by TECFA's students and **TECFA**MOO project team members.

General remark: On X, Mac and Windows client, choose a fixed font!

[Note: this section and subsections are rough drafts]

1.4.1 tkMOO light (Unix, Mac, PC

tkMOO light is a new MOO client with interesting potential, e.g. it provides whiteboards and a graphical desktop. At **TECFA**MOO we so far (June 96) used the whiteboard successfully for experiments.

> "tkMOO-light is a new client which brings mudding kicking and screaming into the early eighties. The client supports a rich graphical user environment, and can be extended to implement a wide range of new tools for accessing M**s.
>
> The client is written entirely in the Tcl programming language and full source code is available in UNIX .tar.gz format and a prebuilt .zip version is available for Windows. The client requires both tk4.1 and tcl7.5 to be installed on your machine. Precompiled binary copies of tk/tcl are available for Windows, NT and Macintosh platforms."
>
> (from http://www.cs.cf.ac.uk/User/Andrew.Wilson/tkMOO-light/

For general documentation about this client, please consult the official "tkMOO light" site. Using the tkmoo client gives you several advantages:

Multi-platform: You can use the same client on Unix, Mac and Windows.

Local editing: You can use local editing (i.e. forget about the in-moo editor). Note that you have to type '@edito +local' to enable "local" editing (same as with emacs clients).

Extensions: You have access to fancier tools, e.g. whiteboards that are integrated with the MOO or "desktops".

Installing tkmoo light is probably overkill for beginners and people who just want to chat. For those purposes, you may want to select some "easier" tool, e.g. Mudwin (PC), Muddweller (Mac) or Tinyfugue (Unix).

Installing tkMOOlite under Win95 [June 96]: We have installed tkMOO-lite 0.2.42 here (e.g. on a 486/66 with 16MB and running Swiss-French Windows 95) and it runs fine, Display is a bit slow. However, it is the only suitable MOO client for building and programming that works under Windows, so get it.

This is one possible installation procedure (it is not too hard, but you should know how to install software, how to use zipped files and how to edit text files).

1. Get tcl-tk ("win41.exe") from SunLabs (http://www.sunlabs.com/research/tcl/install.html[4]) or from our site (ftp://tecfa.unige.ch/pub/software/win/tk-tcl/). It is a self extracting file that will produce installation files.

2. Get the latest tkMOO-lite for Windows distribution from Andrew Wilson's site (http://www.cs.cf.ac.uk/User/Andrew.Wilson/tkMOO-light/Source/[5], i.e. a file called *.zip (e.g. tm0.2.42.zip). Unzip the archive into some directory.

3. Edit the tkMOO-lite application with a text editor, i.e. the file called "source.tcl" and add the location of **this same file** as the first line like in the following example:

   ```
   set tkMOOLibrary "c:/ProgramFiles/tcl-tk/tm0-2-42/"
   ```

   Note that there is also an install script to run under wish, but I found it easiest to proceed as below [D.S].

4. To launch tkMOO-lite you have several alternatives. Here is one using the wish shell: Click on the wish41 application that is in the "bin" directory of the tcl-tk distribution and type: `source "<full pathname of tkMOO-lite>"` like in the following example. And don't forget the quotes around the path and use forward slashes ("/"):

   ```
   source "c:/Program Files/tcl-tk/tm0-2-42/source.tcl"
   ```

   Using wish and "source" is a good way to debug, e.g. it will complain if you entered a wrong file path name in the previous step.

5. A better solution under Win95 is to make a shortcut:

   - Click with the right mouse button on the file source.tcl in the Explorer and make a shortcut, you could name it tkMOO-lite or whatever. You may want to move the shortcut to the main menu.
   - Edit the properties of the shortcut (again using right click), i.e. select "Shortcut" and edit the "Target" field (dunno if those names are correct since I am using a french windoze) to launch tkMOO-lite with wish as in the following example (don't forget the quotes):

     ```
     "c:\Program Files\tcl-tk\Tcl\bin\wish41.exe"  "c:\Program Files\tcl-tk\tm0-2-42\source.
     ```

     Now have fun MOOing, i.e. launch tkMOO-lite by clicking on the shortcut.

6. A third solution is to create a file type association for the extension "*.tcl". Under Win 3.1. you can do this with the File Manager I believe. Under Win95, click on the "My Computer" ("Poste de Travail") icon and go to the "File type" menu. Create a new File Type for TCL (or modify it if you already tried something along this line). Call the new action "open". The command for the action (i.e. the application used) should be something like in the example below. Don't forget the "

[4] http://www.sunlabs.com/research/tcl/install.html
[5] http://www.cs.cf.ac.uk/User/Andrew.Wilson/tkMOO-light/Source/

```
"c:\Program Files\tcl-tk\Tcl\bin\wish41.exe" "%1"
```

This will launch any *.tcl file under wish if you click on it and is therefore the most general solution.

7. Reminder: type '@edito +l' in the MOO to use this client's local editing facility! (@edito -l' will put you back to in-MOO line editing).

Disclaimer: We haven't fully tested this client under Windows (We rather use X servers under Windows and Macs to run the Unix version under Solaris instead... it's easier on installation). Also, while this client is under development consider making frequent updates. You just need to edit the first line of the "source" file.

tkMOOlite@TecfaMOO In order to make optimal use of this client at **TECFAMOO** , do the following in the MOO (on other MOOs, please enquire):

1. Type: '`@addfeature &xmcp_feature`'
2. Optional: '`@playero client=&ansi_tkmoo_client`'
3. Finally, each time you want to use a whiteboard, type: '`@xmcp_challenge`' (or select this command from the tkmoo tools menu). Anyhow, enquire for using whiteboards ....

If you decide to use another client AND you set your @player-options to a special tk-moo client setup, don't forget to change it back:

1. To get the default client, type: '`@playero $client`'
2. you can leave the rest, though if you use a simple client also change back to '@edito -l'.

Also note that tkMOO-light is very new (as of May 96). Currently (July 96) the setup at **TECFAMOO** works with version 0.2.52 but we plan to "stay" in touch with further developments. On Tecfa's Unix machines type: 'tkMOO' to start the client.

1.4.2 Tiny Fugue (Unix)

Runs on BSD or Sys V. Latest version is 3.2beta4. Commonly known as 'tf'. Designed primarily for TinyMUD-style muds, although will run on LPMUDs and Dikus. Features include regexp hilites and gags, auto-login, macros, line editing, screen mode, triggers, cyberportals, logging, file and command uploading, shells, and multiple connects.

"tf" seems to be the client of choice for most MOO programmers (specially those who work over remote connections). TF is easy to install (as far as Unix goes. At Tecfa it runs on Solaris 2.4 (and ran under SunOS 4.1).

Distribution: ftp://ftp.math.okstate.edu/pub/muds/clients/UnixClients/tf/ or ftp://ftp.tcp.com/pub/mud/Clients/tinyfugue/ (latest versions)

Features tf has many advanced features (e.g. a very powerful macro/triggers package) that will not be discussed here.

Ease of Use: Relatively easy, good on-line documentation, type '/help'

Copy/Paste to other applications: ??

Multiple Connections: Yes

Command Repetition: Yes (c-p)

Local Editing: not builtin. However, people wrote macros that can do that (see one of them below).

Session Logs: Yes

Overall judgment: Very powerful client, but learning its advanced features needs some investment.

Local editing with can be done in principle (we did not test it) The following has been taken from the MOO case:

```
/def -mregexp -t"^\\.$" do_localedit = \
        /if /test %{shipping} %;\
           /then /set shipping 0 %;\
           /log -w off %;\
           /sys echo . >> %loced_doc %;\
           /sh %editor %loced_doc %;\
           /quote '%loced_doc %;\
           /sys rm %loced_doc %{loced_doc}\~ %;\
        /endif

; if you want backups, move the document somewhere instead of deleting it

/def -mregexp -t"^#\\$# edit name: (.*) upload: (.*)$" localedit_trigger = \
        /substitute %P2%;\
        /set shipping 1 %;\
        /set loced_doc %{loced_basedir}/tf_edit.doc %;\
        /log -w %loced_doc %;
```

Here is another solution:

```
----------------------------------------------------------------------
>@read 1 on *clients [at river.honors.indiana.edu 8888]

Message 1 on *Clients (\#11431):
Date:     Tue May 23 17:06:27 1995 EST
```

```
From:     Iluvatar (\#1635)
To:       *Clients (\#11431)

; LOCALEDIT.TF --- Local editing support for MOO editors.
;
; By Kirlan@LambdaMOO
; This code is freely distributable.  Please notify the author of
; any major modifications made.
;
; The author assumes no responsibility, period.
; Meaning that I assume no liability for any damages of any sort
; incurred by using this code.
;
; Also meaning that I'll support this code when I feel like it
;
; Simplified version for accounts that can't do terminal multipexing
; Just shells out to the editor
;
/require tools.tf
/set shipping 0
/set editor pico

; set this to an absolute path.  /sys does not do tilde completion
/eval /set loced_basedir %{HOME}

; trigger to finish local editing
; this should really be defined _within_ the start macro but, ugh.
----------------------------------------------------------------------
```

1.4.3 Emacs clients (Unix)

Emacs is a very powerful text editor that allows interfacing to many things including most programming languages, email, News, the WWW and MUDs. If you are unfamiliar with emacs try to talk to someone who uses it and see before you install it. Emacs extensions are written in emacs-lisp. When installing a Mud client you normally have to look at the README file in the distribution and reconfigure some of the *.el files.

The most popular emacs client seems to be mud.el. This client runs on GNU Emacs. Usable for TinyMUD-style muds, LPMUDs, and MOOs. Features include auto-login, macros, logging, cyberportals, screen mode, and it is programmable. Availability: at ftp://parcftp.xerox.com/pub/MOO/clients/ and at ftp://ftp.math.okstate.edu/pub/muds/clients/UnixClients/. See the Mud FAQ list[6] for more emacs clients.

At Tecfa we use Ron Tapia's rmoo client. It is not mentioned in many places but gives best results under X with GNU Emacs.

[6] http://www.math.okstate.edu:80/%7Ejds/mudfaqs.html

The rmoo client

Distribution: (a maybe old version) from TECFA's FTP server[7]. The original version seems to have disappeared from the Net.

When using this client the first thing to do in the MOO is: '@edito +l'. That will ship all editing to emacs. No documentation is available, so to get some help, type 'c-h M' in a RMOO window.

The RMOO client features menus for a few commands (in the original districtuion they only worked for GNU Emacs, i.e. NOT Xemacs). In our slightly modified version the most important commands work from the menu.

Features (Note that we did not test everything in detail, there is no good help)

Ease of Use: Hard if you are unfamiliar with emacs, easy if you are (except for the lack of documentation)

Copy/Paste to other applications: Works

Multiple Connections: Yes

Command Repetition: Yes (M-p)

Local Editing: Yes

Session Logs: Yes (by definition emacs is a text editor after all)

Overall judgment: Very nice client, but hard for people unfamiliar with emacs. Note that learning emacs is a good thing if you work under Unix. E.g. html and latex (this manual was written with) assistants are VERY good.

Available Commands Here are the available commands in RMOO main mode (when just interacting normally with the MOO). The most important thing to do when you work with an emacs clients is to configure your MOO for local editing. On most MOOs, type:

```
@edito +l
```

The most important commands you need to know are the following:

```
Input to the MOO:

RET             rmoo-send

Retrieving previous commands and navigation:

ESC n           rmoo-next-command
ESC p           rmoo-previous-command
C-a             rmoo-beginning-of-line
```

[7] http://tecfa.unige.ch/pub/software/unix/mud-clients/

```
Pasting text:

C-c C-p         rmoo-@paste-kill
                (useful for pasting text in the emacs kill/copy buffer)
C-c C-y         rmoo-send-kill
                (useful for uploading code in the emacs kill/copy buffer)

Quick MOO Mail:

C-c RET         rmoo-mail
```

Here is the full set of emacs bindings, you can ignore most of them (some won't work anythow on **TECFA**MOO):

```
RMOO mode:
Major mode for talking to inferior MOO processes.

Commands:
key             binding
---             -------

menu-bar        Prefix Command
mouse-1         jtext-select-link
ESC             Prefix Command
C-c             Prefix Command
TAB             dabbrev-expand
C-a             rmoo-beginning-of-line
RET             rmoo-send

ESC n           rmoo-next-command
ESC p           rmoo-previous-command

C-c C-p         rmoo-@paste-kill
C-c C-f         rmoo-extras-get-prop
C-c C-v         rmoo-extras-get-verb
C-c C-o         Prefix Command
C-c RET         rmoo-mail
C-c C-w         Prefix Command
C-c C-y         rmoo-send-kill
C-c C-q         rmoo-quit

C-c C-o C-d     rmoo-objects-delete-object-here
C-c C-o C-s     rmoo-objects-write-objects-file
```

```
C-c C-o C-o     rmoo-objects-download-object

C-c C-w C-s     rmoo-worlds-save-worlds-to-file
C-c C-w C-a     rmoo-worlds-add-new-moo
```

Below are the commands active when programming or editing: The most important command is `ctrl-c ctrl-s` which ships back a command to the MOO from an editing buffer. Note: be sure not to delete the "MOO command lines" that pop up when you you start an editing session. In the following example you edit the description of a room:

```
>@edit here.description
```

What you will get in the editing buffer is this:

```
@set-note-text #139.description
You see an modern office, quite standard in appearence.
You see a blackboard on which you can leave a note ('writeb ......').
'Out' will lead you back to the Atrium.
.
```

Don't delete the `@set-note-text #139.description` line NOR the last line (i,e, ".") and insert all editing in between! The exactly same rule applies if you edit a verb, e.g. if you type: `@edit #xxx:my_verb`

```
Major mode for mucking with MOO code.
Commands:
key             binding
---             -------

menu-bar        Prefix Command
C-c             Prefix Command
TAB             rmoo-code-indent-line

C-c C-s         rmoo-upload-buffer-directly  <--- MOST IMPORTANT !
C-c s           rmoo-upload-buffer-directly
C-c ;           rmoo-code-check-semi-colons
C-c "           rmoo-code-insert-quoted-end
C-c C-a         rmoo-code-extras-map

C-c C-a u       rmoo-code-uncommentify
C-c C-a c       rmoo-code-commentify
C-c C-a s       rmoo-code-sin
C-c C-a r       rmoo-code-return
C-c C-a k       rmoo-code-fork
C-c C-a w       rmoo-code-while
C-c C-a f       rmoo-code-for
C-c C-a e       rmoo-code-else
C-c C-a i       rmoo-code-if
```

1.4.4 Muddweller (Mac)

MUDDweller supports multiple sessions, connections with either the communication tool-box or with Mac TCP, a command history and simple file transfer mechanisms. In version 1.2, the main window is now re-sizable and supports more than 32K of word-wrapped text in an arbitrary font and size. This version also adds support for logging a session to a file and for simple macros. Macros can be used to automate the login / logout process or to perform a fixed set of operations when a key is pressed.

The author seems to have given up development at version 1.2. Too bad!

Distribution: at ftp://rudolf.ethz.ch/pub/mud/ and ftp://ftp.tcp.com/pub/mud/Clients/ and ftp://mac.archive.umich.edu/mac/util/comm/ and (a maybe old version) from Tecfa's FTP server[8]

Features

Ease of Use: Easy to use, installation can be a bit tricky for beginners (need to fill in Mac TCP/IP questions)

Copy/Paste to other applications: Works

Multiple Connections: Yes

Command Repetition: Yes (???)

Local Editing: No, means you have to use the mud/moo editors

Session Logs: Yes

Overall judgment: Nice client for ordinary users, not very well suited for development.

1.4.5 MudWin (Windows)

MudWin[9] is a good Windows client that works with major TCP/IP software.

Requirements: You must have a WinSock v1.1 compliant DLL providing TCP/IP access to the Internet. The subdirectory containing WINSOCK.DLL must be in your PATH before starting Windows or MUDWIN.EXE must be placed in the same subdirectory as WINSOCK.DLL.

Distribution: Get it from the MudWin Home Page[10] or (a maybe old version) from TECFA's FTP server[11]

Features

(Note that we did not test everything in detail, but there is a good help, read that if you are interested in things like macro creation (toolbar icons) and automatization of login!)

Ease of Use: Easy to use, also has a configurable icon bar (preconfigured for a few movement and communication commands (in order to pick up an object select first the object in the text window with the mouse then click on the "in-bag" icon

[8] http://tecfa.unige.ch/pub/software/mac/mud-clients/
[9] http://www.microserve.net:80/%7Eserver/mudwin/
[10] http://www.microserve.net:80/%7Eserver/mudwin/
[11] http://tecfa.unige.ch/pub/software/dos/mud-clients/

Copy/Paste to other applications: Works

Multiple Connections: Yes

Command Repetition: Yes (click on the arrow to the right of the input line)

Local Editing: No (?), means you have to use the mud/moo editors

Session Logs: Yes

Overall judgment: Very nice client, but maybe not powerful enough for development work.

1.4.6 Java Applets

Currently, there are several Java Applets that provide basic "chat" functionality. For a list see e.g.: Java MUD Interfaces[12] page. One drawback of those applets are that they need to be installed on the same machine as the MOO runs, so *you* can't install it on your (moo client) machine. Another drawback is that they are not very suitable for construction and development work (as of Oct/96). Get a "real" client like tkmoo-light or an emacs-based client if you work under Unix and know emacs.

At **TECFAMOO** : Check the the **TECFAMOO** pages, e.g. http://tecfa.unige.ch/moo/how_to.html[13] for information on currently installed applets.

1.4.7 [TECFAMOO] Getting a user name

At Tecfa anybody can request a single character for himself by answering to a few questions. You can type '@request' as a guest anywhere in the MOO, but for those who are not familiar with this command, there is a special registration room off the arrival area.

At the **TECFAMOO** Arrival Station, type 'register' to get there. Be prepared to enter:

- A character name, e.g. DanielX
- Your real name
- Your affiliation (institution, school or whatever
- Your interests (e.g. in research, education, etc.)

This information is made available to all users. See the 'finger <name>' command.

New characters at **TECFAMOO** will get a certain amount of quota (currently 25000 bytes and 10 objects) and the ability to build things. Programmer bits have to be requested from a wizard.

In case of trouble: Please contact a wizard (Type 'help wizard-list' for a list of wizards).

1.5 Manners on a MOO

See the "Manners" section in the The evolving TecfaMOO Book Part I: Concepts[14] (use 'back' on your client to come back here !

---

[12] http://www.cold.org/javamud/
[13] http://tecfa.unige.ch/moo/how_to.html
[14] http://tecfa.unige.ch/moo/book1/tm.html

1.6 Bugs, please help us !

TECFAMOO is first of all a MOO for research and for researchers. Since it is constantly modified and since our resources are scarce expect to stumble upon bugs sometimes. We would be very glad if you could report them to us. Ideally you should use the bug reporting form:

```
Usage:    @bug-report <bug report title>
          g.g.
    @bug-report @go is broken
```

this form will ask you a few question and optionnally will allow you to automatically include your last error traceback. So use this form as soon as you get a traceback please. Thanx!

Chapter 2

Communication and Navigation

2.1 Basic standard MOO commands

Note that “standard” means that chances are very high that those commands will be available on a given MOO. They may not for the following reasons:

- The Moo is based on a very different core
- The commands are disabled for your player class. E.g. often commands like @join and @go are disabled for guests.

2.1.1 Basic Communication

MOOs posses a sophisticated set of communication commands. The basic commands are shared among MOOs.

In order to list connected players:

```
@who    Lists  connected persons
@who    <name> lists connect time information for a given person
```

Within a MOO: type 'help communication' for more help on this topic.

'say' - Say anything out “loud” to everybody in a room 'say' is probably the command most frequently used. Note however, that on most MOOs you can find a “stage talk” feature that allows you to address yourself directly to a person.

```
Syntax:    say <text>
           " <text>

    <text> = any kind of text

Exemples:
```

```
  >"hello
  You say, "hello"
  >say this is wonderful
  You say, "this is wonderful"
```

Others will see for example:

```
  Daniel says "hello"
  Daniel says "this is wonderful"
```

'emote' - Non-verbal expressions

```
Syntax:      emote <text>
             : <text>
```

Exemples:

```
  >emote smiles
  Kaspar smiles
  >:laughs
  Kaspar laughs
  >:thinks it is time to go home
  Kaspar thinks it is time to go home
```

'whisper' - private communication

```
Syntax:     whisper "<text>" to <user>

     <user> must be somebody in the same room
```

Note: You must type the quotes around your text, also: the page command has more or less the same effect. On some MOOs people in the room can see that you whisper something to somebody.

Examples:

```
  >whisper "now this is a secret" to Daniel
  You whisper, "now this is a secret" to Daniel.
```

The other will see something like:

```
  Kaspar whispers, "now this is a secret"
```

'page' - Communication over distance 'page' is being used to send messages in real time to users which are not in the same room. Note that standard information displaying the effects of this command are often different from MOO to MOO. Some users customize this command very much, so don't be surprised !

```
Syntax:     page <user> <text>


Examples:

  >page MooBoy Hi, have you got a moment ?
  Your message has been sent.

The other will see:

  You sense that Daniel is looking for you in Daniel's Office.
  He pages, "Hi, have you got a moment ?"
```

Now look at a more customized version of that pageing mechanism:

```
  >page MooBoy Hi, have you got a moment ?
  Your message has been sent.
  >

The other person will see:

  Kaspar [at Daniel's Office] tells his big dog to head for your place.
  He pages, "Hi, have you got a moment ?"

And when he pages back:

  >page Kaspar sure !
  The big dog gently grabs the message and heads for Kaspar
  >
```

2.1.2 Basic Navigation

One way of navigation is simply to "walk" through exits by typing the exit's name. Note that one some Moos you can only do that (besides using more exotic artefacts like vehicles).

The following commands will also work on most MOOs:

"go" will allow you to cross several rooms at the same time, or to cross an exit that has a strange name, e.g.:

```
go n,s,down

go help
```

"@go" is mostly implemented as teleportation command. Often it works only if you enter the room object number (e.g. #1111) or if you remembered the room. Most Lambda-based cores have an @addroom command (type 'help @addroom') for remembering rooms. At **TECFAMOO** we use the nickname facility to remember all sorts of objects (see section 2.5.2).

```
@go <room number>
    e.g.
@go #1111

@go <remembered room name>
@go mooseum
```

"@join" will allow you to join a user at another location. Please page him if it is ok to join if you feel that the person is in some private area. You can also use the knock command for this as in the following example:

```
>@knock kaspar
You have knocked on the walls of Kaspar [on&off]'s location.
Kaspar [on&off] invites you to @join him in The Irradiation Chamber.
>@join kaspar
You join Kaspar [on&off].
The Irradiation Chamber
A dark chamber lit by a warm, orangish glow. The light in here
seems to almost be alive.  It seems like it's breathing.
Daniel is sitting in the chair.
You see Box 'O Generics, vs, and Web-Thing here.
Irradiate (out on her feet) and Kaspar [on&off] are here.
Looks like you can go north, door, south, and west from here.
```

Some MOOs also implement walking algorithm that will find a path to a location. At **TECFAMOO** you can try walk to <location>. Even more fun are various transportation artifacts, like trains, cars and so on. Look at those objects and read the description or the help if needed!

2.2 MOO Mail and Mailing Lists

"MOO Mail" includes two functionalities:

1. The MOO email system allows you to send and receive messages to and from other users.
2. It allows the creation of mailing-lists. Each MOO has a set of public mailing lists to which may subscribe. If you are a frequent user of A MOO, subscribe yourself to the most important ones !

The most important commandes you should know are the following:

- '@nn' will display all new messages (both in your personal mailbox and new messages on mailing lists you have subscribed.

- `'@send'` will allow you to send messages.

```
@send Kaspar
      will put you in the mailer (or pop up a local editing window if
      you client can handle it)
@send *General
      will allow you to send a message to the *general mailing list.
```

- `@mail` will list messages (default is last 20 messages)

```
@mail
     lists messages on your personal mail-box
@mail on *general
     will list messages on the *general mailing list
@mail 1-20
     will list the first messages 1-20 on your mail box
@mail 30-$ on *general
     will list messages 30 to last on the mailing list *general
```

Help is available on the following commands:

```
@mail     -- seeing a table of contents for a collection of email messages
@read     -- reading individual messages
@next     -- reading the 'next'     message
@prev     -- reading the 'previous' message

@send     -- composing and sending a message to other players
@answer   -- replying to one of the messages in a collection
@forward  -- resending one of the messages in a collection somewhere else

@rmmail   -- discarding some subset of a collection
@unrmmail -- undoing the most recent @rmm on a collection
@renumber -- renumbering the messages in a collection
@keep-mail - marking messages in a collection as exempt from expiration
```

For viewing collections other from your own, the following commands are useful:

```
@rn          -- list those collections that have new messages on them
@subscribe   -- indicate that you want @rn to report on a given collection
                  and add yourself to its .mail_notify list
@skip        -- ignore any remaining new messages in a given collection
@unsubscribe -- ignore a given collection entirely from now on
                  and remove yourself from its .mail_notify list
@unsubscribed-- show the mailing lists that you aren't subscribed to.
```

```
@subscribed  -- like @rn, but shows all lists that you are subscribed to
                even if they have no new activity
```

2.3 [TECFAMOO] Janus' Channels System

Channel Systems are non-standard and may vary from MOO to MOO. At Tecfa we have 2 channel systems (xymox's Channel FO #122) and Janus' new more versatile Channel FO (#3619).

Channel Systems allow people to "chat" over the same channel while in different rooms. They can be compared to CB or IRC channels in some ways with the difference that you can listen to several channels at the same time. Also, since channels are at the same time rooms, you can also @go to the "channel room". Things spoken there will be shown to everybody listening on a channel.

Some channels are public, others may be private. Special channels may be used to convey system information (e.g. connections) to the users.

Within the MOO you can get more information about the Channel System by typing: 'help &channel_feature'. It will give you a command summary.

Currently (May 96), there are 3 public channels. Stay tuned for other channels.

```
&public_channel -- general chat channel
&help_channel   -- channel dedicated to help people
&prog_channel   -- the place were to ask your prog questions
```

There are two "one way" channels (you can only listen to them):

```
&checkpoint_channel -- announce when a checkpoint start,
                       finish, if it succeded, ...
&connection_channel -- announce when people login/logout
```

2.3.1 Adding the Channel System

First check, if you don't allready "have" channels by typing: **@features**. If you see the `#3619 channel feature` like this:

```
>@features
Feature Name
------- ----
#225    xy's improved social FO
         [.......]
#3619   channel feature
------- ----
10 features found.
```

you don't need to read further. If the Channel feature is missing on your character, you do this:

```
@add-feature &channel_feature
```

Casual users probably want to use just existing channels and don't need to read further but should remember the 'help &channel_feature' command. Actually, `'help <most anything>'` will most of the time bring you a step further in the MOO.

2.3.2 Channel Selection

To list all open channels and some statistics, type `@xch'` or `@xchannels`. You will see something like:

```
>@xch
LT  Channel     Owners  Users  Online   LT  Channel     Owners  Users  Online
--  ----------  ------  -----  ------   --  ----------  ------  -----  ------
!!  Admin       ~Janus  7      3        --  Help        ~Eval   8      2
-!  Checkpoint  ~Janus  7      1        --  Prog        ~Eval   8      2
-!  Connection  ~Janus  6      2        --  Public      ~Eval   11     3
```

To add yourself to the listeners of a channel, type:

```
@xlisten <channel>
    e.g.
@xlisten help
```

To set a channel as default channel for talking, type:

```
x>><channel>
  e.g.
x>>public
```

Now to remove yourself from a channel or switch default channels you also have the following commands, but remember at least the `@disc <channel>` command.

```
@xco*nnect <channel>       - Quit your curr. ch.,
                             and set your curr. ch. to <channel>
@xdis*connect [<channel>]  - Quit listening to all channels [only <channel>].
@xonly <channel>           - Quit all channels except <channel>.
```

To see who is listening to a channel use `@xwho`. To list all subscribed listeners (i.e. disconnected people) use `@xwho-all`.

```
@xwho [<channel>]       - Shows online listeners on <channel>
                         (curr ch. by default)
@xwho-all [<channel>]   - Shows all listeners on <channel>
                         (curr ch. by default)
```

2.3.3 Communication over channels:

Communicating on the default channel The most basic command is 'x' like in the following example:

```
>x anybody working on this ?
[+][Admin] You ask, "anybody working on this ?"
```

Here is a more complete set of commands.

```
x <text>                - say <text> on your current channel
x:<text>                - emote <text> on your current channel
xmo <text>              - as above
xto <player> <text>     - directed speech ([to player] <text>) on your curr. ch.
x-<player> <text>       - as above
x`<player> <text>       - as above
xth <text>              - . o O (<text>) on your current channel
```

Switching default and communicating to another channel:

```
x>><channel> [<text>]   - set your current channel to <channel> [and say <text>]
xmo>><channel> [<text>] - as above [and emote <text>]
xto>><channel> [<player> <text>] - as above [and direct speech]
xth>><channel> [<text>] - as above [and think <text>]
```

Communicating to another channel (without switching default):

```
x//<channel> <text>     - say <text> on <channel> (rather than on your curr. ch.)
xmo//<channel> <text>   - emote <text> on <channel> (as above)
xto//<channel> <player> <text> - direct speech on <channel> (as above)
xth//<channel> <text>   - think <text> on <channel> (as abobe)
```

Other commands may be added in the future.

2.3.4 creating new channels

To make your own channel, just make a kid of the channel class (i.e. the generic channel). See also the in-MOO help[1].

```
@create &channel_class named <whatever you want>
```

This also allows you to make a conference room that is also a channel, so that people that need to be in another room can listen or participate to the conference (or to more than one conference simultaneously). But don't `@chparent` a complex room to `&channel_class`, you will use its features, unless you change its generic too.

The audience to a given channel can easily be restricted and so allow to make channels dedicted to some projects with only the members of the project being able to talk and/or listen to the channel.

To allow people/objects to use your channel through the &channel_feature:

```
@trust &channel_feature to manage <OBJ your_channel>
```

To allow people/objects to announce (talk) on your channel:

```
@trust <class> to announce <OBJ your_channel>
```

To allow people/objects to listen to your channel:

```
@trust <class> to listen <OBJ your_channel>
```

Examples:

a) to make a public channel named my_public_channel, 3 steps:

```
1) @trust &channel_feature to manage my_public_channel
2) @trust $everything to announce my_public_channel
3) @trust $everything to listen my_public_channel
```

b) to make a channel, named my_semi_public_channel, where all players can listen, but where guests can't talk (though guests can listen), steps:

```
1) @trust &channel_feature to manage my_semi_public_channel
2) @trust players to announce my_semi_public_channel
3) @distrust $guest to announce my_semi_public_channel
4) @trust players to listen my_semi_public_channel
```

c) to make a private channel, named `Tims_friends` where only a selected few can talk and listen, type first `@trust &channel_feature to manage my_public_channel`, then for each player do:

```
1) @trust ~<player> to announce Tims_friends
2) @trust ~<player> to listen Tims_friends
```

If you can't find your channel, list all kids of `&channel_class`:

```
@kids &channel_class
```

[1] http://tecfamoo.unige.ch:7777/help/&channel_class

To list permission on your channel, type:

```
@list-l <your channel>
```

More on ownership permissions can be found in section 5.2.

There are a few parameters you can customize if you wish:

```
@channel_prefix &channel_class is "[$perm][$title] "
@channel_title &channel_class isn't set.
@channel_color &channel_class is "%green%bold"
```

The channel prefix is (as its name suggest it) the prefix that is put at the beginning of every line of text displayed by the channel to its listeners. The permissions of the listeners will be displayed instead of `$perms` and the title of the channel (or its name if no channel title is set) will be displayed instead of $title.

The listeners perms are:

- + if the channel is the one the listeners is also currently talking on.
- - if the listeners is just listening and not talking on the channel.

So if you see:

```
[-][Public] <some text>
```

you know that if you want to say something on Public you will have to switch to the channel first or to make a comment to that channel (with `x>>public <test> or x//public <test>`

The channel color is not currently used.

2.4 Social Features

Social Features play an important role in MOO communities. [more to come, in the meantime type @help features in the MOO]

To learn about available features you can for example look at an experienced user at your MOO and type @features for `<user>`.

2.5 Remembering and finding things

Several MOOs have specialized data bases for finding things of interest. Most have at least facilities for remembering rooms (you can give names to rooms you visited and teleport there once remembered thos names). [more needed here].

2.5.1 Remembering Rooms

A lot of MOOs have the `@addroom` command implemented. At **TECFA**MOO we don't, see section 2.5.2 on Nicknames. Here is the help from Diversity University MOO:

```
Usage:  @addroom [<room>] [as <name>]

Adds a room to your list of known rooms (this allows you to reference it by
name with such commands as @go and @move).  If <room> is not specified, your
current room is assumed.  If you specify "as <name>", <name> will be used as
the name for the room in your room database, instead of the room's actual
name (this allows you to specify abbreviations and such).

This command can be abbreviated down to '@addr'.

Example:  @addr #11 as SU
 ..would add room #11 (the Student Union) to your personal database
 of rooms under the name of "SU".

See Also:
  @go      - Teleport yourself to a specified location.
  @move    - Teleport another object to a specified location.
  @rmroom  - Remove a room from your personal list of rooms.
  @rooms   - Display all rooms known by name.
```

2.5.2 [TECFAMOO] Nicknames

Nicknames allow you to give nicknames to objects. Unlike aliases, nicknames are only valid for you and you can add nicknames to objects that you don't own. Nicknames should be also be matched before regular names and aliases.

The following commands are obsolete at **TECFA**MOO and are going to be deleted: @rooms, @addroom, and @rmroom; if you had a db of rooms, it has been converted to nicknames.

Basic commands needed:

```
@nickname*s <object> - displays the nicknames assignes to an object
@nickname            - displays the usage
@nickname everything - list all the nicknames you assigned

@addnickname*s <nickname> [, <nickname>, ..., <nickname>] to <object>
@rmnickname*s  <nickname> [, <nickname>, ..., <nickname>] from <object>
```

Example:

```
@nickname
Usage:  @nicknames <object>    => list the nicknames you assigned to <object>.
        @nicknames everything  => list all the nicknames you assigned and the associated objects.
>@nickname me
You have not assigned any nicknames to MooBoy [offline] (#1324) <~MooBoy>
>@nickname everything
Foyer (#258) has the nickname Foyer.
```

```
>@addnickname stag to #3522
Now The Old Stag (#3522) has the nickname stag.
>@nicknames everything
Foyer (#258) has the nickname Foyer.
The Old Stag (#3522) has the nickname stag.
>@go stag
The Old Stag
 [.....]
>@nickname here
The Old Stag (#3522) has the nickname stag.
```

Part II

EXTENDING THE WORLD (Building)

Chapter 3

Introduction to Building and Construction

In most MOOs standard players (users) are allowed to build a limited amount of objects (see chapter 5). Usually people will start building a "home" in MOO terminology, e.g. an office, a virtual house or any other place which will be your favorite private place on the MOO. Therefore, we first will learn how to build rooms and exits in section 3.2. But let's hear some additional information before we start.

On-line help on building is pretty well done in the usual MOO. If you type 'help building'[1] you will see something like the following:

```
>help building
There are a number of commands available to players for building new
parts of the MOO.  Help on them is available under the following topics:

creation -- making, unmaking, and listing your rooms, exits, and other objects
topology -- making and listing the connections between rooms and exits
descriptions -- setting the names and descriptive texts for new objects
locking -- controlling use of and access to your objects
```

Most of the following stuff is built on the internal MOO documentation.

3.1 General information on building things

It is useful to know what building implies at a more technical level, so you should read the next paragraph. However if it sounds to technical for you you may skip most information and come back to it later. However, you must know how to list and find things you own, so read the paragraph after the next one.

[1] http://tecfa.unige.ch:7777/help/building

What does "building" mean? Building relies on the MOO's object-based architecture. When you create an object for your personal use you create a "child" of a generic object (a class) that allready exists. E.g. digging a room implies making a child of the object called `$room`. Except when you build rooms and exits you usually use the `@create` command to build objects (see section 3.3 for details). Most of those objects can be customized as you will learn later. Usually you are least expected to add a description.

Most MOOs do have some place where generic objects are on display. E.g. at Tecfa we have a MOOseum[2] (`#258`). Other MOOs have implemented commands that will list the most important generic objects. Else, enquire!

The last important thing to know is the object naming and numbering scheme. Each objects on the MOO (including yourself) is an object that has a number. People with builders or programming permits can refer to objects as numbers, e.g. `#84`. Numbers are not always very practical, therefore there exist other ways to refer yourself to objects. In each standard MOO, each objects has:

- A name (which can be changed with the `@rename` command

Usually you can only refer to objects by name if you are in the same room, or if you use some special commands that will go and find a name in some database. Take the `@who` command as an example. E.g. `'@who Kaspar'` will look up the "kaspar" string in the players data base and return some information. **TECFAMOO** users also should have a look at section 2.5 on nicknames that allows you to specify your own aliases for any object you want to remember.

- Very important generic objects on MOOs are corified, i.e. you can refer to them with a name preceeded by the `$` sign, like `$room`.

Note that programmers never should directly refer to core objects by numbers. It will make their core much more portable. At **TECFAMOO** (this is not standard) you have in addition the following conventions:

- You can refer to users (in programs) by using a `~` like in `~fred` (only programmers need to know this)
- Programmers can insert their own objects into the local core database and those objects are referred to with a `&`, like for instance `&channel_class`. Programmers, please local corify your stuff here!

Listing and killing objects you own Before you start building lot's of things it is a good idea to learn how you can keep track of them and how you delete them. First remember that any object you create (except rooms and exits) will be on you (in your inventory). Remember the 'inv' verb:

```
inv      Lists all objects you are currently carrying
```

Here are a few useful verbs that help you keeping track of things (they should be available in most MOOs). Note that most `''@'' verbs` can take arguments for doing more sophisticated queries. Use 'help' in order to know more.

[2] http://tecfamoo.unige.ch:7777/258

```
@audit  Lists all objects in your possession.
        Audit is the most important verb you need to remember.

@prospectus
        like @audit, but lists more information

@quota  Lists available quota
@count  Counts the number of objects in your possession

@location <object>
        will list the location of an object
@move <object> to here
        will move an object to the room you are in
```

At **TECFAMOO** we also have a `Short Audit (#1419)` feature. Il will give a shorter listing than `@audit` or `@prospectus` if you type:

```
@sp
```

Type `@addfeature #1419` to add this feature for your usage.

3.2 Rooms and exits

Building nice rooms with furniture and customized features and customized exits can be quite challenging for a beginner, but building a few connected rooms is not too hard as you will see soon.

Now before you start building lots of rooms, remember this:

- Don't start building lots of rooms the first time you start mooing. Most people will ignore your own personal "tiny scenery" unless it is there for some purpose. E.g. at **TECFAMOO** you can visit Eric's "Ticino Village"[3]. If you want to visit, start from his office (`#397`). Anyhow you have limited quota on most MOOs and they will be used up pretty fast.
- You CAN NOT CONNECT to any other room you don't own! So something like `@dig exit|in to #11` wont work if you don't own `#11`. Ask a wizard or an administrator to link your rooms with the rest of the universe. Note that most MOOs do have an explicit building policy which you should respect. E.g. at **TECFAMOO** go to the "Underground Construction Office" (`#483`) near the entrance aread and read the documentation.

3.2.1 Basic digging

Creating rooms and exits with @dig There are several commands for building rooms and exits, of which you need to know at least `@dig`. Here is the definition from the help system of **TECFAMOO** :

[3] http://tecfamoo.unige.ch:7777/446

```
>help @dig
Syntax:  @dig "<new-room-name>"
         @dig <exit-spec> to "<new-room-name>"
         @dig <exit-spec> to <old-room-object-number>

This is the basic building tool.  The first form of the command
creates a new room with the given name.  The new room is not connected
to anywhere else; it is floating in limbo.  The @dig command tells you
its object number, though, so you can use the @move command to get
there easily.

The second form of the command not only creates the room, but one or
two exits linking your current location to (and possibly from) the new
room.  An <exit-spec> has one of the following two forms:

        <names>
        <names>|<names>

where the first form is used when you only want to create one exit,
from your current room to the new room, and the second form when you
also want an exit back, from the new room to your current room.  In
any case, the <names> piece is just a list of names for the exit,
separated by commas; these are the names of the commands players can
type to use the exit.  It is usually a good idea to include explicitly
the standard abbreviations for direction names (e.g., 'n' for 'north',
'se' for 'southeast', etc.).  DO NOT put spaces in the names of exits;
they are useless in MOO.

The third form of the command is just like the second form except that
no new room is created; you instead specify by object number the other
room to/from which the new exits will connect.

NOTE: You must own the room at one end or the other of the exits you
create.  If you own both, everything is hunky-dorey.  If you own only
one end, then after creating the exits you should write down their
object numbers.  You must then get the owner of the other room to use
@add-exit and @add-entrance to link your new exits to their room.

Examples:
    @dig "The Conservatory"

creates a new room named "The Conservatory" and prints out its object
number.

    @dig north,n to "The North Pole"
```

```
creates a new room and also an exit linking the player's current
location to the new room; players would say either 'north' or 'n' to
get from here to the new room.  No way to get back from that room is
created.

    @dig west,w|east,e,out to "The Department of Auto-Musicology"

creates a new room and two exits, one taking players from here to the
new room (via the commands 'west' or 'w') and one taking them from the
new room to here (via 'east', 'e', or 'out').

    @dig up,u to #7164

creates an exit leading from the player's current room to #7164, which
must be an existing room.
```

Common mistakes and hints: Beginners often do the following: They create a a room like "Snow Castle" with a command like:

```
>@dig enter|out to Snow Castle
Snow Castle (#7164) created
......
```

Next they create another room

```
 @dig garden|office to Lovely Garden
```

Next they go to the second room created ("Lovely Garden" here) and try to dig an exit into the first room by typing:

```
 @dig enter|out to Snow Castle
Snow Castle (#8545) created
```

This will create a second room called "Snow Castle". If you want to dig into a room that **exists already**, you have to use room object numbers like this:

```
  @dig enter|out to #7164
```

Also, don't use exit names containing more than one single word because the `@dig` command AND the users will get confused if you don't type it right. Here is an example that demonstrates the principle:

```
>@dig drown under | "swimm to shore" to sea
Usage:  @dig <new-room-name>
    or  @dig <exit-description> to <new-room-name-or-old-room-object-number>

>@dig "drown under" | "swimm to shore" to sea
```

```
Usage:  @dig <new-room-name>
    or  @dig <exit-description> to <new-room-name-or-old-room-object-number>

>@dig "drown under"|"swimm to shore" to sea
sea (#3951) created.
Exit from beach (#1442) to sea (#3951) via {"drown under"}
 created with id #1780.
Exit from sea (#3951) to beach (#1442) via {"swimm to shore"}
 created with id #4346.

>drown under
I don't understand that.
>go drown under
You can't go that way (drown).
>go "drown under"
sea
You see nothing special.
Obvious Exits: Swimm to shore (to beach).
```

3.2.2 Dealing with exits

For simple room and exit creation, the `@dig` verb should be enough (so can skip reading this is you are in a hurry). However, there are situations where you need to know a few additional commands, e.g. when you want to remove an exit or an entrance or when 2 people want to link rooms they own. In the latter case you can use the @dig command to create and exit and an entrace, but `@dig` will complain. Don't worry, but write down the numbers of the created exits and ask the owner of the other room to add them with the `@add-entrance` `@add-exit` commands (see below) Here is an example:

```
>@dig door|back to #1111

Exit from island (#1429) to TecfaMOO ARRIVAL area (#1111) via {"door"}
created with id #4194.
However, I couldn't add #4194 as a legal entrance to TecfaMOO ARRIVAL
area.  You may have to get its owner, Daniel to add it for you.

Exit to island (#1429) via {"back"} created with id #3997.  However, I
couldn't add #3997 as a legal exit from TecfaMOO ARRIVAL area.  Get
its owner, Daniel to add it for you.
```

So the exit you have created in **your room** must be added as an **entrance** to the other room and the entrance you have created from the other room leading back to **your room** must be added as an **exit** to the other room! **Warning: Dealing with exits can be very tricky!** Don't just `@dig` lots of rooms and exits without really checking what happend, else you may wind up with many exits that are in nowhere space ! Also be very careful when you recycle rooms. Double check before you hit return and recycle a room that you don't want to destroy. If it happens you need a programmer bit to restore all exits and entrances to connected rooms or else recycle them too and rebuild everything!

The following information on exit commands is straight from the MOOs help system:

```
help @exits
Syntax:  @exits

Prints a list of all conventional exits from the current room (but
only if you own the room).  A conventional exit is one that can be
used simply by typing its name, like 'east'.

>help @entrances
Syntax:  @entrances

Prints a list of all recognized entrances to the current room (but
only if you own the room).  A recognized entrance is one whose use is
not considered to be teleportation.
```

Note that sometimes you can't list exits and entrances of rooms you don't own.

```
help @add-exit
Syntax:  @add-exit <exit-object-number>

Add the exit with the given object number as a conventional exit from
the current room (that is, an exit that can be invoked simply by
typing its name, like 'east').  Usually, @dig does this for you, but
it doesn't if you don't own the room in question.  Instead, it tells
you the object number of the new exit and you have to find the owner
of the room and get them to use the @add-exit command to link it up.

>help @add-entrance
Syntax:  @add-entrance <exit-object-number>

Add the exit with the given object number as a recognized entrance to
the current room (that is, one whose use is not considered
teleportation).  Usually, @dig does this for you, but it doesn't if
you don't own the room in question.  Instead, it tells you the object
number of the new exit and you have to find the owner of the room and
get them to use the @add-entrance command to link it up.

>help @remove-exit
Syntax:  @remove-exit <exit>

Remove the specified exit from the current exits list of the room.
Exit may be either the name or object number of an exit from this
room.
```

```
>help @remove-entrance
Syntax:  @remove-entrance <entrance>

Remove the specified entrance from the current entrances list of the
room.  Entrance may be either the name or object number of an entrance
to this room.
```

3.2.3 Customizing your room

Assume that you are the proud owner of at least one room.

Declaring a home One thing you may want to do right now is declaring one of those rooms as your "home". Note that you can only "live" in a room you own or in a room where its owner has added you to the list of potential residents.

```
@sethome
```

will tell the MOO that the current room is your home, meaning that:

- you will wake up in this room once you connect
- the 'home' verb will teleport you to this place from any place in the room.

Describing your home Next, you want to describe each room you own. You can do this in three ways:

1. If you have simple MOO client (like Muddweller, Mudwin, tf, a Java applet) beginners should use the `@describe command`. If you are in the room, you simply would type:

   ```
   @describe here as ......

   example: @describe here as "A lovely park with thousands of pink flowers"
   ```

2. If you have a simple client, type `@notedit here` or `@notedit #xxxx` (if you are not inside the room) and learn how to use the built-in MOO editor, e.g. type `'look here'` once you are inside the MOO editor.
3. If you have a serious MOO client like an (emacs mode or tkmoo lite) type `@edit here`, type in your description and ship back the result to the MOO. Read the documentation of your client and tell the MOO to use "local editing" by typing `'@editoptions +local'`. See sections 11.1.1 and 1.4 for more details on how to edit things!

3.2.4 Advanced building

Advanced building bascially means 2 things, you can:

1. Customize messages on rooms and exits, to make actions more lively

2. Change the "parent" of your room (which is `$room` on most rooms into a more fancy kind of room. If you are inside the room you want to change, type: @chparent here to `#xxx`. See chapter 4 for details on some fancy available rooms at **TECFAMOO** . We suggest that beginner should start using Ken's Generic Classroom (not just for building virtual classrooms). It is available in most "serious" MOOs and relativly easy to customize (see section 4.2).

Customizing messages (ok this section needs some better writing, I know!!)

By customizing messages (not just on rooms and exits) you can create your scenery much more "lively". Here is an example:

```
>g17
 You stumble over some open boxes into Daniel Schneider's Office
```

Other persons in the room would have seen:

```
 MooBoy makes a shrieking sound when he enters office #17
```

To learn more about messages, consult the internal MOO help, e.g. type:

```
help @messages
help messages

container-messages
          the messages on objects that can contain other objects
exit-messages
          the messages on exit objects
thing-messages
          the messages on objects that can be taken and dropped
```

Pronouns and other substitutions Since help about using pronouns is quite long, we include the in-MOO help about that topic here. Pronouns are useful for referring in more natural way to people in messages. You can use pronouns in a lot of situations, e.g. to customize messages on yourself, on your rooms and on exits.

```
>help pronouns

Some kinds of messages are not printed directly to players; they are
allowed to contain special characters marking places to include the
appropriate pronoun for some player.  For example, a builder might
have a doorway that's very short, so that people have to crawl to get
through it.  When they do so, the builder wants a little message like
this to be printed:

        Balthazar crawls through the little doorway, bruising his knee.

The problem is the use of 'his' in the message; what if the player in
```

```
question is female?  The correct setting of the 'oleave' message on
that doorway is as follows:

        "crawls through the little doorway, bruising %p knee."

The '%p' in the message will be replaced by either 'his', 'her', or 'its',
depending upon the gender of the player.

As it happens, you can also refer to elements of the command line
(e.g., direct and indirect objects) the object issuing the message,
and the location where this is all happening.  In addition one can
refer to arbitrary string properties on these objects, or get the
object numbers themselves.

The complete set of substitutions is as follows:

        %% => `%'  (just in case you actually want to talk about percentages).
    Names:
        %n => the player
        %t => this object (i.e., the object issuing the message,... usually)
        %d => the direct object from the command line
        %i => the indirect object from the command line
        %l => the location of the player
    Pronouns:
        %s => subject pronoun:          either `he',  `she', or `it'
        %o => object pronoun:           either `him', `her', or `it'
        %p => posessive pronoun (adj):  either `his', `her', or `its'
        %q => posessive pronoun (noun): either `his', `hers', or `its'
        %r => reflexive pronoun:  either `himself', `herself', or `itself'
    General properties:
        %(foo) => player.foo
        %[tfoo], %[dfoo], %[ifoo], %[lfoo]
               => this.foo, dobj.foo, iobj.foo, and player.location.foo
    Object numbers:
        %#  => player's object number
        %[#t], %[#d], %[#i], %[#l]
           => object numbers for this, direct obj, indirect obj, and location.

In addition there is a set of capitalized substitutions for use at the
beginning of sentences.  These are, respectively,

   %N, %T, %D, %I, %L for object names,
   %S, %O, %P, %Q, %R for pronouns, and
   %(Foo), %[dFoo] (== %[Dfoo] == %[DFoo]),... for general properties

Note: there is a special exception for player .name's which are assumed to
```

```
already be capitalized as desired.

There may be situations where the standard algorithm, i.e., upcasing
the first letter, yields something incorrect, in which case a
"capitalization" for a particular string property can be specified
explicitly.  If your object has a ".foo" property that is like this,
you need merely add a ".fooc" (in general .(propertyname+"c"))
specifying the correct capitalization.  This will also work for player
.name's if you want to specify a capitalization that is different from
your usual .name

Example:
Rog makes a hand-grenade with a customizable explode message.
Suppose someone sets grenade.explode_msg to:

  "%N(%#) drops %t on %p foot.  %T explodes.
   %L is engulfed in flames."

If the current location happens to be #3443 ("yduJ's Hairdressing Salon"),
the resulting substitution may produce, eg.,

  "Rog(#4292) drops grenade on his foot.  Grenade explodes.
   YduJ's Hairdressing Salon is engulfed in flames."

which contains an incorrect capitalization.
yduJ may remedy this by setting #3443.namec="yduJ's Hairdressing Salon".

Note for programmers:
 In programs, use $string_utils:pronoun_sub().
 %n actually calls player:title() while %(name) refers to player.name directly.
```

3.3 Construction with @create

The general principle for building other things than rooms is quite simple, you just create a "kid" of a generic object with the `@create` command. Let's have a look at this command as it appears when you use the internal help system:

```
Syntax:  @create <class-name> named "<names>"
         @create <parent-object> named "<names>"

The main command for creating objects other than rooms and exits (for
them, see 'help @dig'; it's much more convenient).

The first argument specifies the 'parent' of the new object: loosely
```

```
speaking, the 'kind' of object you're creating.  <class-name> is one
of the four standard classes of objects: $note, $letter, $thing, or
$container.  As time goes on, more 'standard classes' may be added.
If the parent you have in mind for your new object isn't one of these,
you may use the parent's name (if it's in the same room as you) or
else its object number (e.g., #4562).

You may use "called" instead of "named" in this command, if you wish.

An object must be fertile to be used as a parent-class.  See help
@chmod for details.

The <names> are given in the same format as in the @rename command:
        <name-and-alias>,<alias>,...,<alias> [preferred]

See 'help @rename' for a discussion of the difference between a name
and an alias.
```

Here is a simple example:

```
@create $thing named "A little thing",truc
```

This will create a child of the simple `$thing` generic object, called "A little thing" with the alias "truc". The `$thing` object is not very useful for building something, but programmers often start with this. Go and visit the MOOseum if you are interested in owning fancier objects or look at the parents of objects you meet in the MOO and create a child for yourself:

```
@parents <object>

e.g. @parents here
     @parents me
     @parents #139
```

Usually, most objects have a "help" on them that will explain how to customize it. Encourage your programmer friends to write helps!

Chapter 4

[TECFAMOO] Building things at TECFAMOO

For more introduction to building in MOOs, please look elsewhere for the moment:

- consult the The Cow Ate My Brain Article[1], also available at the same place via ftp.
- Type 'help building' !!
- Wade through the mailing lists on each MOO, e.g. *new and *www at **TECFAMOO** . People annonce new generic objects you can use there ! (See section 2.2) for information on how to read MOO Mail.
- Check out the MOOseum (see section 4.1)
- You may also read the LambdaCore User's Manual[2] since our MOO is lambda core base at it's origin, but be aware: this manual is old and a few things may be different!

4.1 The MOOseum

The MOOseum (#258) is the most important resource center for builders at **TECFAMOO** . Within the MOO, go and check it out !!

4.2 Fancy Rooms

TECFAMOO has several fancy rooms, most of which can be found in the MOOseum. In order to create a fancy room, you best do the following:

1. @dig ...... (sorry no information YET about digging above)
2. `@chparent <this room> to #1862`

[1] http://tecfa.unige.ch/pub/documentation/MUD/MOO/The-Cow-ate-my-Brain.text
[2] http://tecfa.unige.ch/cgi-bin/info2www?(LambdaCoreManual.info)

4.2.1 Ken's generic Classroom

Generic classrooms are quite versatile and can be used for other purposes than classrooms, e.g. they work well for offices (just remove the teachers desk). Following information is the same as you get in the moo, by typing:

```
  help #374
     - or -
  help here      (inside a generic classroom)
```

```
>help #374
Generic Classroom (#374):
----
                        How To Use This Room
     To create new tables, desks, shelves, bulletin boards etc. use the
'@addfurniture' command. You can 'look', 'sit', or 'put' things on most
objects you create. While sitting you are heard only by others sitting
with you. If you 'stand' or 'speakup' everyone can hear you. 'Look' at
objects to see what's on them.
GENERAL
   look blackboard       sit Big Desk             @status
   writeb Hi There!      stand                    put map on Bulletin
   eraseb 4              speakup I think that..   get map
   to Ken Hi there!      look clock               look Big Desk
SPECIAL
   cleanb                  - erase entire blackboard
   register/unregister Ken  - add/rm Ken from *current* class
   @addfurn/@rmfurn Desk    - add/remove a piece of describable furniture
   door open/closed         - restrict entry to persons in *current* class
   @mkclass/@rmclass Math   - add 'Math' to list of classes taught here
   @setup Math              - makes 'Math' the *current* class
   @stifle on/off           - disable seated persons emoting to everyone
   @sign on/off             - show 'this.session_msg' at @who command
   @authorize/@unauth Ken   - add/rm Ken from list of authorized users
   @restrictions on/off     - restrict Special verbs to authorized users
   @tutorial                - view a short tutorial on using classroom
   @fix                     - correct seating mixups if they occur
```

4.2.2 Ria's multi-room

Following information is more or less the same as the one you get by typing `'help #1862` in the MOO.

Introducing The Generic Outdoor Secured Place-Oriented(seats included) Room With Enhanced Integrating Descriptions, Exit Matching, Security and Details. Take a look... Not only can it have seats, but it can also have tables, hooks, beds, hangers, boxes, bags, vases, bowls, etc, etc, etc... It has 4 ways of securing the room, integrating descriptions (where an object blends in with the description instead of saying that 'You see ___ here.' N,No,Nor,Nort and North would all match, unlike the Generic Room, where you must type out the full name or alias. You can also add details to the room, without wasting quota to make an object just to 'look' at.

Help for Generic Secure Outdoor Place-Oriented Room with Exit matching, details and integrated descriptions:

```
----------------------------------------------------------------------
@add-place <place>           Adds a table, hook, chair (or anything else)
                             to the room.
@put <place> is <message>    Re-sets the message when someone puts an
                             object in the place.
@take <place> is <message>   Re-sets the message when someone takes an
                             object from the place.
@take-verb <place> is <verb> Adds a verb that you type in order to take
                             an object from <place>.
@rmtake-verb <place> is <verb>Removes verb from those that you type in
                             order to take an object from <place>.
@put-verb <place> is <verb>  Adds a verb that you type in order to put
                             and object in <place>.
@rmput-verb <place> in <verb> Removes verb from those that you type in
                             order to put an object in <place>.
@alias <place> is <alias>    Adds an alias to <place>.
@rmalias <place> from <alias> Removes an alias from <place>.
@capacity <place> is <amount> Sets the amount of objects <place> can hold
@describe-place <place>      Allows you to change the description of
                             <place>.
@type <place> is <type>      Sets the kind of objects that may go at
                             <place>  ... ie, if it's a hook, write the
                             number for the generic plant.  If it's a
                             chair, use $player.
@visible <place> is <message> Re-sets the empty-place message.  Use 0 if
                             the place should be invisible when empty.
@remove-place <place>        Deletes a place.
@places                      Shows all places and what they can hold
----------------------------------------------------------------------
@integrate <object> is <msg> Whenever <object> is in the room, <msg>
                             is displayed.  Use "%i" in place of the
                             object's name.
@unintegrate <object>        Reverses effects of @integrate
----------------------------------------------------------------------
@secure [<type>]             Secures the room based on <type>
```

```
                              For help, exclude the <type>
@accepting player | object    Accepting depends on the player moving
                              the object OR the object itself.
@accept <person>              Adds <player> to the accept list.
@deny <person>                Adds <player> to the deny list.
@unaccept <person>            Removes <player> from the accept list.
@undeny <person>              Removes <player> from the deny list.
@invite <person>              Invites a person to enter the room for
                              1 hour.
---------------------------------------------------------------------
@detail <detail> as <descrip> Adds a detail to the room.
@rmdetail <detail>            Removes a detail
---------------------------------------------------------------------
@summer...                    Sets the message to appear during
@winter...                    the appropraite season.  According
@spring...                    to the time of day.  The time may
@fall <time> is <message>     be any of: sunrise, morning, afternoon,
                              evening, sunset or night.
---------------------------------------------------------------------
@add-noise <noise>            Adds the noise <noise>.
@rm-noise <num>               Removes the noise indexed by <num>.
@noisiness <num>              Changes how noisy the place is (1-10).
@noises                       Lists the noises defined.
---------------------------------------------------------------------
@roominfo/@status [<place>]   Shows information about the room or a
                              specific place.
```

4.2.3 The Tutorial Room

Programmed and ported to **TECFAMOO** by DaveM@DU. Thanx a lot ! Tutorial Rooms can be used for writing linear text in several parts in-MOO.

[Note we should rather write a WWW interface to this]

```
>help here
Underground Construction Office (#483):
----
                       Tutorial Room
                      Basic Usage help

This room simulates a walk through tutorial without really leaving the room.
The basic commands are simple.

   'go'          -  to start a listed text module.
   'q', 'quit'   -  to quit a tutorial once it is started.
```

```
For more detailed help, type 'go help' in the room for a built-in tutorial.

Inside the text module:

   'n', 'next'   -  while in a text module will move to the next page.
   'p', 'prev'   -  while in a text module will move to the previous page.

   'l', 'look'   -  while in a text module will show the current page's text.

   '@mailme here' - will send a copy of current page to your email address
          [NOTE: You must have a registered character to use @mailme]

   'speak speakup su speak-up speak_up' to communicate with someone outside
                 of the module you are in.

With that basic knowledge you can use the room.

Take any questions or problems to DaveM. Writers of text modules should
see the help on the Generic Text Module (#502) as well as 'go help.'

>go help
```

HTHelp: Introduction

```
Welcome to a very specialized room.

It might be helpful to picture a room with various text reading
or display modules scattered around it. Each module can hold one
or many people. When inside a module you are surrounded by the
pages of the display text that is installed.

Through simple commands you can move your way through the text in
relative privacy. Or a group can be together and discuss without
fear of bothering other module users.

Currently this room simulates a linear walk-through tutorial.
Development of a simple hypertext interface is soon to be finished.
Send any questions or comments to DaveM.
```

HTHelp: Getting Started

```
When you enter the room, you will be told what "Text Modules" are
```

installed. To enter one is easy.

Type "go"

If there is only one text module installed you will enter it. If more than one is present you will be shown the list and asked which module to enter.

Or, you can type "go" followed by the module's name. Just like you typed "go help" to start this tutorial. This instruction might also be written as "Type 'go <module name>'" With the <> brackets telling you to substitute the module name. You do not type the brackets or the quotes.

HTHelp: Quiting

The second most important thing to know about something is how to stop. With the text modules, this too is easy.

Type "q" or "quit".

This command will return you to the main tutorial room.

HTHelp: Watching

There is another way to enter a text module.

Let's say that you and a friend or two want to view a text module together. Maybe it is an assignment, or someone wants to discuss some text with you. You can enter a text module as a group. The steps (I hope) are also easy.

- One person types "go <module name>"
- Everyone else types "watch <first person's name>"

Now it will be true that where one goes, the others will follow. Anyone in a group can change the page. Also, inside a text module your speech is limited to that text module. (More on that later).

HTHelp: Moving around

When you are inside a text module you will want to change pages, and maybe even look at the page again, especially if your discussion gets hot and heavy.

These are the commands you need to know.

```
   'n' or 'next'   -  moves to the next page.
   'p' or 'prev'   -  moves to the previous page.

   'l' or 'look'   -  will show you the current page's text.
```

See the help menu below? These basic commands will get you far in using this room.

HTHelp: Communicating

As I mentioned before speech is restricted for people inside a text module. So commands like "say" and "emote" (" and :) will only be heard by those who are reading with you.

You will be able to see and hear people who are not in a text module. To talk outside of your text module you have two commands:

```
   "speak <text of message>" - announce a message to others
                 (synonyms: speakup su speak-up speak_up)

and "` to <person> <message>" - whispers a private message so
                                only <person> hears it.
```

HTHelp: Use Summary

```
   "go"   -  to enter a text module
   "q"    -  to leave a text module
   "w <person>" - to join <person> in a text module.

   "n", "next" - to proceed to next page.
   "p", "prev" - to go back to previous page.

   "l", "look" - to see current page's text again.

   "speak"  -  to talk with someone ouside of your text module.
```

```
   "` to <person> <message>" - to whisper a message to anyone in
                               the room.
```

That's all you need to know to use this room. You can type "q" now and use the room. However if you are interested in how to use the room to develop teaching resources, type "n".

HTHelp: Developing

The Generic Hypertext/Tutorial Room (#501) is a specialized room for the presentation of text materials. It is currently able to go through text in a linear fashion. Presenting one page at a time and going from beginning to end.

I am currently working on phase II of this room's development, which will allow a simple hypertext interface that will allow users, with the typing of a single word, to follow a branch of a menu tree, or skip to a particular topic of interest, or even follow a footnote. It could even be used for people to create their own story line. But, alas, this is yet to be.

The following pages will help you see how to use the two parent objects, in creating your own tutorial rooms.

HTHelp: Two Objects?

I wonder if you caught that phrase in the last line. You need at least two objects to use this room.

First you need to create, or have access to a child of the Generic Hypertext/Tutorial Room (#501). I have one off of my room. To make it, I followed these three steps.

```
     @dig tutor|out called "DaveM's Tutorial Room"
     tutor                     ; to enter the room
     @chparent here to #501
```

[If you don't understand any of those commands see the help related to them.]

This gives you the user interface, much like a vcr gives you access to the information of a videotape.

HTHelp: Text Modules

```
As you stand in the room, you may notice that there are no text
modules listed for the room. This tutorial is always available (as
simple as "go help"). But any other text modules need to be created,
filled and registered by you (or others).

The information for the text modules is stored on a child of the
Generic Hypertext/Tutorial Text Module (#502).

     @create #502 called "Tutor Help"

will create a module that when registered in the room will be called
"Tutor Help", but for now you have it in your inventory, just
waiting to be defined and filled.
```

HTHelp: Adding text

```
To add a page of text to the module you need to issue this command:

   add <module>

You will then be asked to specify a title for the page. This is the
second part of what appears within the [brackets] in the title line.
For example, this page is entitled "Adding Text" on the module
called "Tutor Help". Do you see where they are used?

After that you can enter in the text of the page, one line at a
time. When you have finished, type a period on a line by itself.
This tells the program that the page entry is finished.

Note: if you want to insert a page, add it first, then use edit, change order.
```

HTHelp: Formatting text

```
In writing your text, keep a couple things in mind.

  1. The Room will add nothing except a title and a menu bar to your
text. No added spaces or anything else. I added the line of dashes
below the title. You can if you want. The full responsibility of the
presentation of text is yours.
```

```
  2. Consider the resources and experience of your readers in mind.
I think it is wise to assume that people don't have a wonderful
client and they don't have either @linelength or @pagelength set. I
think to aim at 70 character lines and 18 line pages is reasonable
and considerate. You set your own standards, but remember your
audience.

Summary: What you type is what people will see.
[Enough preaching, on with the tutorial.]
```

HTHelp: Working Offline

```
I like to do a lot of my work off line and then "import" them into
the right objects. I don't know about your client or your
communication software, but if you do have a way to use ASCII upload
or the "dumping" of text into the Moo it can make some tasks easier.

If you like to do that, the line format for adding text pages is
this:

 line 1: add <module name>
 line 2: Page Title
 line 3: First line of entered text
 line 4: Second line of entered text.  (repeat)
 line n: .
```

HTHelp: Making Changes

```
Once you enter a page of text it is inevitable that you will want to
change something. There are two basic types of changes to make:

To change the ordering of pages or to remove a page, use -
       reorder <module>

To edit the text of a page, use -
       @notedit <module>

With both types of changes, you will first be shown a list of current pages
defined for the module. You must first choose the number of the page you
want to make changes for. Then you will proceed to the next step.
```

HTHelp: Change Order and Delete

```
This value represents where in the ordering of pages a particular
one will be shown. This is the order that is always shown in the
index or listing of contents.

To change it, you simply enter the new place for the chosen page.

Delete a page:
------------------------------------

This choice will remove the page from the module.
```

HTHelp: Editing Text

```
Editing a page within the text module is as easy as editing anything
on the moo. If you understand how to use the Moo editor, or if you logon
to the moo through a client that is capable of local editing (like emacs
mud.el) then you should have little problem editing your text.

To edit text, type '@notedit <module>'.

After choosing the page you wish to edit you will proceed to your editor.
Make whatever changes you want to make. When you are finished with your
editing, type 'save' (or whatever command saves your local editor). The
page you just edited will be restored to its proper place and you can
then proceed to quit the editor. Once you 'save' the text you cannot save it
again without restarting the editor. A bookmark is kept to remind the editor
where the page has to be stored when it is saved. This bookmark is cleared
with the save command.
```

HTHelp: Registering Modules

```
Okay, so you have a text module and you have filled it with pages of
text. Before it can be used in a room it must be registered. Go into
the room with the module in your inventory and type:
      register <module>
You will be told that it is registered and you will be able to "go"
into it. If you are finished with a module go into the room and type
"remove <module>".
```

```
After this, you can put the module wherever you wish. You can keep
it in your inventory or put it in a closet or a \$container. I would
recommend not putting it in the room, it might be confusing.

Finally, don't forget to "@describe <module> as <text>". People in
the room can "look" at the registered modules to see the description
and also a table of contents.
```

HTHelp: History and Future

```
This room is an offshoot of two things: the Generic Notice Board
(#250) and Donald. Donald asked if a variation of my Notice Board
could be made as simple tutorials. As we talked, this room (which is
more then either of us imagined it might be) has come about.

Phase I is in its final stages of development. This is an emulation
of a walk-through tutorial. But instead of using a room per page, it
uses one object for the room and one object for all the text of one
tutorial.

Phase II will be the addition of hypertext capabilities. It will not
be as full blown as World Wide Web Hypertext, but it will offer the
possibility of following other links beside next page and previous
page.

I hope you enjoy these object. Talk to DaveM if you have any
comments, questions or bugs. Thank you.
```

4.3 Ken's Turing Robot

Let's look at the bot Ken ported over to **TECFAMOO** :

```
You see a Turing Robot designed to interact in vigorous 'Eliza-like'
conversation with other folks in its vicinity. It has key words, sentence
patterns, random responses, and question responses - all user programmable.
It has some ability to recognize where it is and to whom it is talking.
Type @exam botname to see all the available commands. For detailed assistance
in programming this bot type 'help botname'. For a discussion of the issues
involved in testing machine intelligence see Turing's paper 'Computing
Machinery and Intelligence'.  To start up the bot, drop it, activate it,
and say 'hi'. Please report bugs to ken/cdr@CollegeTown.
```

You also can look at it via our WWW interface.[3].

[3] http://tecfamoo.unige.ch:7777/1684

Basic operation

This is a slightly edited version from 'help bot'[4]. [NOTE: should fix this woo interface to help some day]

1. ADDING A WORD FOR YOUR BOT TO RESPOND TO:
   To see what words your bot already responds to type `'seewords botname'`. To teach your bot to respond to 'donut' with either 'I like donuts too.' or 'Donuts are very tasty!' just type '@addword botname' and enter the keyword 'donut'. Then enter the appropriate responses a line at a time. End with a single period on a line by itself.

2. ADDING A PATTERN FOR YOUR BOT TO RESPOND TO:
   Suppose you wished your bot to hear something like

   ```
       MY DONUT ISN'T VERY TASTY
   and respond with
       WHAT'S SO GREAT ABOUT A TASTY DONUT?
   To do this you must teach your bot to respond to the pattern
       MY a ISN'T VERY b.
   ```

   To understand what patterns look like, type `'seepat botname'` and study the examples. For additional assistance on understanding the syntax of patterns type `'help regular'`. When you think you are ready to add a pattern type

   ```
   @addpat botname' and enter the following line when asked to do so:
       my %(%w*%) isn't very %(.*%)
    Then type in the response form:
       What's so great about a %2 %1?
   ```

   Add as many response forms as you wish on separate lines. End with a period on a single line.

3. ADDING RANDOM RESPONSES:
   These responses are triggered whenever your bot can't find a keyword, a pattern, or a question. To see the responses already programmed type:
   `'seerandoms botname'`.
   To add a new random response type:
   `'addrandom botname'` and enter a new response.

4. ADDING A RANDOM RESPONSE TO A QUESTION:
   When your bot senses a question is being asked it responds with a random 'answer'. To see the random question responses already programmed into your bot type `'seequestionresponses botname'`. To add a new response type `'addquestionresponse botname'`.

5. To remove words, patterns, random responses or answers use the appropriate 'rm-' command. For example, to remove pattern number 5 on your bot just type 'rmpat 5 from bot'.

[4] http://tecfamoo.unige.ch:7777/1684?web_verb=help

6. Your bot responds to patterns in the order in which it matches them. If you wish to move pattern number 7 to a position nearer the beginning of the list you might type something like `'mvpat 7 on botname'`. You will then be asked to enter a line number to move pattern to.

7. CAUTION: This bot was designed for serious educational and experimental purposes. It makes an excellent 'guide' or 'tutor' and is an interesting vehicle for the study of the limits of language understanding using an 'Eliza' approach. BUT.. it has great spam potential since it responds to nearly everything it hears. 'Hush botname' when not needed and please be considerate of others around you ..

   Please forward comments and bugs to Ken Schweller, Ken/CDR @COLLEGETOWN

Chapter 5

Quota and Security

5.1 [TECFAMOO]The Quota System

As of May 1996, we use a double object and byte quota system. When you need more quota, ask an administrator and he will give you more under the condition that you work on a serious project. "Tiny scenery" builders usually don't fall under this category anymore since our MOO is growing.

Type '@quota' to see your allocated quota, example:

```
>@quota
MooBoy has a total building quota of 500,000 bytes for 150 objects.
His total usage is 324,082 bytes for 80 objects.
MooBoy may create up to 175,918 more bytes of properties or verbs.
```

This means that MOOBoy can use up 500K and build 150 objects (that includes exits). (Note that this is a lot)

Quota policy:

1. Standard builders and programmers have 25K bytes and 10 objects of quota.
2. People doing interesting projects can have more.

Administrators: To change quota consult: 'help @set-quota'.

5.2 [TECFAMOO] Security - Trust and Distrust

(really under construction !!!)

The "Trust" system offers you more flexibility and allows collaborative work. The thre essential commands are:

```
@trust <source> to <event> <destination> [for <duration>]
```

```
@distrust <source> to <event> <destination> [for <duration>]
@trust-list <destination>
```

First `@trust-list <destination>` will show you the list of events supported by `<destination>` as well as which classes are trusted/distrusted for each event. The term “class” means any object, or generic object, or special class (see below).

Example:

```
>@trust-l me

Trust -- Janus (#11) <~Janus>
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

build
=====
Classes trusted to build object can describe the object, set the messages, changes the
  +Trusted   : owners

mail
====
Classes trusted to mail the user.
  +Trusted   : everything.

move
====
Classes trusted to move the object.
  +Trusted   : everything.

[... and so on ...]
```

This means means that everyone can mail me. But that only I can describe me.

Now the list of trusted/distrusted classes can be changed with the @trust and @distrust commands.

```
Example:

@distrust $everything to move me
(Now no one but me can move myself.)

@distrust $guest to mail me
(Now everything but guests can send me mail.)

@trust ~Charlotte to build me
(Now Charlotte can change my description, name, ...)
```

As you can see a class can be various things... a generic (`$guest`), a special class (`$everything`) or an object/player (`~Charlotte`).

So far the special classes are:

```
$everything  => objects/generics/special classes... everything!
admins       => administrators
owners       => players that own <destination>
programmers  => objects with a prog bit
players      => objects with a player bit
```

The trust/distrust system allow you to have more controls over "who can do what" on your stuff. Right now it offers you controls about who can enter your rooms, move your objects, send you mail, page you, build with you...

More to come. Adding new events is fairly easy. Please let me know what events you would like to be supported and what event scope you would like to be extended...

As always... comments/suggestions/... are welcome ! Janus

WARNING I strongly suggest you never:

```
@trust <whoever> to OWN <whatever> ***
```

The own event will be usefull for multiple ownership when the sytem will be open to the public... meanwhile, don't use it. If you use the OWN event, do it at your own risk and don't whine someone recycled your stuff...

All the other events are ok :)

P.S. the @refuse commands have been deleted, use @distrust instead (works the same, but it is easier to maintain from a programmer point of view).

5.3 [TECFAMOO] Multi-Ownership

(really under construction !!!, this section needs will be done once the system is fully tested out)

See also section 9.4 on Multiownership for programmers!!

Part III

INTRODUCTION TO MOO PROGRAMMING

Chapter 6

Basic MOO Programming

Note that you also ought to look at the same time at chapter 11 which explains how to use the available editing and development tools. You also want to read section 7.1 which gives a short introduction into how a MOO system operates.

6.1 Documentation on MOO Programming

1. Get the official MOO programming manuel from one of the archives (e.g. from
2. The Ftp archive at Xerox[1] or use the html version, available in several places, e.g.:
   - LambdaMOO Programmer's Manual[2] (at Xerox)
   - LambdaMOO Programmer's Manual[3] (in the "lost" MOO library)
   - LambdaMOO Programmer's Manual[4] (ccs.neu.edu)
   - LambdaMOO Programmer's Manual (at TECFA, our place)
3. The Lambda core user's and programmer's manuals are also useful and can be found at in the Xerox archives[5]. Note however, that most MOOs have their core modified and that a few use a REALLY different core. At Tecfa you consult them on-line: LambdaCore User's Manual[6] and Programmer's Manual[7], but again: they are outdated even for LambdaMOO.
4. Also, look for various cheatsheets and other useful stuff: Most can be found in serveral places:
   - Jerome P. McDonough' MOO Library[8]
   - The Ftp archive at Xerox[9], check also the /contrib/docs subdirectory[10].

[1] ftp://ftp.parc.xerox.com/pub/MOO/
[2] ftp://ftp.parc.xerox.com/pub/MOO/ProgrammersManual.texinfo_toc.html
[3] http://lucien.berkeley.edu/MOO/ProgrammersManual.texinfo_toc.html
[4] http://www.ccs.neu.edu/home/ivan/moo/lm_toc.html
[5] ftp://ftp.parc.xerox.com/pub/MOO/contrib/docs/
[6] http://tecfa.unige.ch/cgi-bin/info2www?(LambdaCoreManual.info)
[7] http://tecfa.unige.ch/cgi-bin/info2www?(LambdaProgMan.info)Top
[8] http://lucien.berkeley.edu/moo.html
[9] ftp://ftp.parc.xerox.com/pub/MOO/
[10] ftp://ftp.parc.xerox.com/pub/MOO/contrib/docs/

- At Tecfa we keep a few of those documents around in an FTP directory

Note, however that this documentation is pretty useless for total beginners, unless you are an experienced programmer. But it will become usefull pretty rapidly, so GET IT!

Then, get the various cheatsheets available:

Available Tutorials:

- Check out the tutorials below. Note however that they are being created right now (end of october 1995). Excecept for the section 6.3 which introduces both to programming and MOO programming, they have not been user tested.
- Start with Canton Becker's Tutorial[11] - (local copy)[12]
- The Bouncing Ball tutorial[13] is another nice alternative.
- Also do the jduJ's Duck Tutorial[14] - (local copy)[15]. Or/and look at an alternative Duck Tutorial[16] - (local copy)[17]
- Many Moos have Tutorials on-line. E.g. see Irradiate's MOOseum on **TECFAMOO** (telnet://tecfamoo.unige.ch:7777[18]). The MOOseum has some 'tape-based tutorial' which you you should be able to find on any MOO (ask!).
- Most MOOs have one or several lists that help people with programming: At **TECFAMOO** look at the *Tutorials and the *prog-tips mailing list.

Note that you also ought to look at the same time at chapter 11 which explains how to use the available editing and development tools.

6.2 Programming: the General idea

You can read this section, before or after section going through either a on-line programming tutorial in the MOO (e.g. the CDR tapes) or the one in section 6.3.

[Don't laugh, this is under construction!]

6.2.1 What is programming?

Programming is instructing a computer to do something for you with the help of a programming language. The role of a programming language can be described in two ways:

1. Technical: It is a means for instructing a Computer to perform Tasks

[11] http://www4.ncsu.edu/unity/users/a/asdamick/pub/moo/tutorial.txt
[12] http://tecfa.unige.ch/pub/documentation/MUD/MOO/Becker-Tutorial.text
[13] http://www.ccsi.com/ cscottw/moo/moo.html
[14] http://lucien.berkeley.edu/MOO/ducktutorial.txt
[15] http://tecfa.unige.ch/pub/documentation/MUD/MOO/jduj-duck-tutorial.text
[16] http://lucien.berkeley.edu/MOO/ProgrammerTutorial
[17] http://tecfa.unige.ch/pub/documentation/MUD/MOO/duck-tutorial.text
[18] telnet://tecfamoo.unige.ch:7777

2. Conceptual: It is a framework within which we organize our ideas about things and processes.

According to the last statement, in programming we deal with two kind of things:

- data, representing "objects" we want to manipulate
- procedures, i.e. "descriptions" or "rules" that define how to manipulate data.

According to Abelson and Sussman ([ABELSON, 1985, 4])

> ".....we should pay particular attention to the means that the language provides for combinign simple ideas to form more complex ideas. Every language has three mechanisms for accomplishing this:
> *primitive expressions*, which represent the simplest entities with which the language is concerned
> *means of combination*, by which compound expressions are built from simple ones, and
> *means of abstraction*, by which compound objects can be named and manipulated as units."

A programming lanugage should both provide means to describe primitive data and procedures and means to combine and abstract those into more complex ones.

The distinction between data and procedures is not that clear cut. In many programming languages, procedures can be passed as data (to be applied to "real" data) and sometimes processed like "ordinary" data. Conversely "ordinary" data can be turned into procedures by an evaluation mecanism.

6.2.2 The Elements of Programming

There are very few essential elements one needs for programming:

1. variables (things to store information), i.e. local variables and object's properties in the MOO language.
2. procedural abstractions (named collections of instructions), i.e verbs in the MOO language
3. branching (IF's)
4. side-effects ("Set", "="), to assign values to variables
5. procedure/function calls
6. loops (as "cheap" alternative to procedure calls)

Those major elements rely on a number of things like:

- primitive data structures, e.g. numbers and symbols
- input/output functions
- libraries of useful procedures and functions

To make a programming language useful, you also need things like:

- Ways to build compound data structures (e.g. lists, arrays, records, objects)
- Complex conditions and several kinds of loops
- Error handling mechanisms

Many programming languages have special features that make them useful for a given task, e.g. the MOO language has:

- A sophisticated permission system
- Threads (forks for "parallel" execution of code)
- Ticks (for controlling execution of forks)
- User management functions
- etc.

In some sense the MOO Programming language can be compared to an operation system. A server manages users and programs in somewhat similar way.

6.2.3 Object Oriented Programming

In a loose sense, Object Oriented Programming is based on the idea that one deals with a community of communicating objects. "Pure" OO languages are based on a quite strong "message passing" metaphor, i.e. computation is the result of sending messages between objects following clear protocols. From a more local point of view, OO programming means that each object can send, receive and handle a specific set of messages. Messages are handled by so-called methods which do look like ordinary procedures, but can only alter the internal state of an object or send messages to objects.

Now what is object oriented in LambdaMOO ?

1. All data is stored in objects
2. Objects have single names (the #xxx number in the MOO db). Each Object has therefore a single identity.
3. Objects maintain internal state (that is the information contained in the slots)
4. Objects can inherit values (default property values) and methods (verbs) from other objects. This is in contrast to most OO Languages, where inheritance is disciplined by a subclass relation between abstract "classes" that do not function as actual objects - called "instances". Inheritance in the MOO obeys a simpler scheme, prototype inheritance, where chosen objects (called "Generic Objects" in the MOO) serve as models for other objects. Generic objects are almost the same as their instances.
5. Inheritance is single, meaning MOO objects can only inherit from a single Generic Object. However, "feature objects" do add some flexibility to this scheme.

6. Internal states of objects (properties) can be accesses from the outside without any special protocol (with restrictions, see permissions). That is not "pure OO style" either.

So the MOO language is object-based but not really really object-oriented in the strong sense. This does no mean that one can't program relatively "pure" OO style, but it certainly is not enforced.

[more to come]

6.2.4 Functional Programming

[more to come]

Functional programming means calling that functions with some arguments. Those functions return a value, a list of values or multiple values. In most programming languages, function calls are done in the following way: function-name (argument1, argument2, ...)

Function calls often used in an embedded way, like outer-function(next-function(next-function(innermost-function(args)))) Here is a imaginary example:

```
display-nicely
   (sort-list
          (retrieve-all-players(player-data-base, guest-data-base)
          :how alphabetically)
    )
```

Functional programming style is useful in several situations, e.g.

- When you want to implement filters, i.e. procedures that that transform data into temporary data-structures you need for some job.
- In complex "systems" where the application layer allows application programmers to specify how certain operations on certain objects are to be done. A good example is TECFA's WWW interface. E.g. a function like :htext has the task to return a list of "good" html which then can be dumped back to a www client by another procedure. And you can define your own :htext for any MOO object you own.

It is possible to mix functional and procedural and object oriented programming style. However this is not always a good idea. Ideally procedures "sold" as functions should just return a value and NOT modify permanent data-structures. This way others and even you can use your functions and *just* expect that they will take some data, do something with them and return the result back to your procedure for for further use.

If function do alter data somewhere, you have to document that somewhere.

6.2.5 Procedural/structural Programming

[more to come]

Hierarchically well organized algorithms work on relatively simple data structures. Associated with procedural/structual programming is often "top-down" development of programs.

6.3 MOO tutorial, Level 0 ("holder")

The MOO programming language is object-based (and object-oriented in the sense that permanent data and procedures are all attached to objects). So if you want to program anything, you foremost deal with objects. MOO programming is a bit different from ordinary programming as you interact directly with the MOO database. Everything you program will be stored there. In this sense you can consider your MOO client as a data base interface.

This will be shown in a simple example, a tool that will allow you:

1. to store short bits of information
2. to edit those bits
3. to show those bits to other people

The product you will program has just educational purpose, but actually might want to use it for real. Carry it around and use it to remember important things to do in the MOO for example. The following transcript shows some interactions with a a tool that allready has been programmed. DON'T try out those commands now, since you first have to program something like this first ! Note that commands entered after the > are entered by the user, the rest is feedback from the MOO.

```
----------------------------------------------------------------------
>help holder
holder (#2108):
----
This is an exercice left for the reader!
Type @examine in the meantime....

>@examine holder
holder (#2108) is owned by MooBoy (#1324).
Aliases:  holder and memo
Information Holder
Key:  (None.)
Obvious Verbs:
  add <anything> to holder
  dump holder
  show <anything> on holder
  clear holder
  erase <anything> from holder
  g*et/t*ake holder
  d*rop/th*row holder
  gi*ve/ha*nd holder to <anything>

>dump holder

>add Improve MOO tutorial
I don't understand that.
```

```
>add Improve MOO tutorial to holder
Inserted ''Improve MOO tutorial'' to holder

>dump holder
1: Improve MOO tutorial

>add changer le dump verb (feedback si le holder est vide) to holder
Inserted ''changer le dump verb (feedback si le holder est vide)'' to holder

>show 2 on holder
MooBoy shows item 2 on holder: changer le dump verb
feedback si le holder est vide)

>add test to holder
Inserted ''test'' to holder

>dump holder
1: Improve MOO tutorial
2: changer le dump verb (feedback si le holder est vide)
3: test

>erase 3 from holder
You erased element 3: "test" from holder

>clear holder
Do you really want to erase all information bits? [Enter 'yes' or 'no']
>no
OK, information onholder was too important for erasing
----------------------------------------------------------------------
```

Remember the goals we want to achieve in order to understand why we are programming all the following features. The tutorial in this section will **take some time** (count four hours at least and **more** if you never did any programming before). Also note that you maybe will not understand everything in a first go. To really understand what you did you should add new functionalities to this objects once are done with the tutorial (or else program something else). So let's start:

6.3.1 Creating an object

Let's assume that you don't want to modify the behavior of an existing object but work on a new one. The first thing you need to do is to create an object and give it a name. We we call this object "holder" because it holds information. Choose any other name if you don't like it. However, for ease of use go for a simple name, not something like "My favorite ToDo List tool".

You can create an object from any **fertile generic object**. Generic objects are some kind prototypes containing a set of properties (some having default values) and a set of verbs. If you don't want to profit from the already existing features of a more sophisticated object you make a child of $thing (aka #5): Now type:

```
@create $thing named holder
```

The MOO will reply with something like:

```
You now have holder with object number #2108
and parent generic thing (#5).
```

By typeing 'inv' you can see that you carry this new object. Keep it on you or drop it to the floor (but don't forget to pick it up when you walk away). If you loose track of this object '@audit' will always show you what you own and you can retrieve it by typing '`@move <object> to me`'. If the object is not with you or in the same room you have to refer to it by its number (e.g. #2108). Each object has a unique number.

Now as we said, this object has already a few inherited properties and verbs. Some of those properties you bf own and you can modify their values. You don't own the inherited verbs, but you may bf use them. If you are curious about details you may inspect your object from several angles (see section 11.2 on how to do that. For now let's just describe your object (something you should know how to do), i.e. type:

```
@describe holder as Information Holder
```

Now you can look at your object as the following transcript shows:

```
>look holder
Information Holder
```

Not a very good description and you may want to change that sometimes!

6.3.2 Creating a property "holding"

Since we want to remember bits of information we now need a place to store them, i.e. a property (slot). Remember that all permanent information is stored in object slots and see section 7.3.1 for more details on property creation: Now, let's create a property (slot) on our new object for that. We name it 'holding' because it will hold some information (this is an arbitrary decision and we could have chosen any other name). Type:

```
@property holder.holding {}
```

This created a property 'holding' with the initial value of "{}", i.e. an empty**list**. Lists are quite practical and flexible data-structures as you will see later. For now just take it for a fact, that we will use the list attached to the "holding" property in order to store bits of information. Type '@d holding.' if you want to be sure that you really added the property you wanted to the object.

Note, that we *could* have simply typed:

```
@property holder.holding
```

This would have created a property with an initial value of '0'. [19] But it is always preferable to initialize a property with the kind of value (see section 7.5.1, page 111) you later want to attach to it, otherwise some verb may break, e.g. you couldn't possible add a number to a string.

In case you forgot to set as initial value as the empty list "{}" you are not lost: Just set the value of the property like this:

```
@set holder.holding {}
```

6.3.3 The first verb "add":

Now let's create a verb that will allow you to add an information bit to this object. There are basically 2 uses for verbs. Either one uses them as user commands or as procedures called by other procedures (or user commands). User commands are a more tricky matter because you have to specify verb argument types that will allow the command parser to work correctly. Here we basically will create a verb called "add" that could be used in the following way (don't enter that command now, it won't work yet!)

```
>add "The MOO is great" to holder
```

You may now have a glance at section 7.4 on page 108 to see how command parsing works. Otherwise do that later and just type in the following without understanding details. It is important to type "@verb ...." **only once for each given verb** you want to program. Else you will wind up with several verb definitions and the MOO (and yourself) will get confused! In case you want to kill a verb (including its definition) type `@rmverb <your object>:verb`, e.g. `@rmverb holder:add` in this case. Now, proceed with entering the verb definition:

```
>@verb holder:add any to this

Verb added (0).
```

What we did here was to define a **verb argument specification** for a verb called "add" that will be attached to the "holder" object. The verb will be activated if it encounters a command like 'add "the moo is great" to holder'. The "any" keyword means that any kind of input could be entered. The "to" keyword means that the "to" preposition has to typed. The "this" keyword means that the indirect object should match the object to which the verb belongs (or as you will learn later some generic object from which an object inherits the verb).

Now let's program the verb itself. Note that we assume here that you know how to use the in-MOO text editor, or that you use a client that can ship editing to a local editor (see section 1.4). In the latter case, make sure to have local editing set. ('@edito +l will do that).

Now type `'@edit holder:add'`.

(1) If (and only if) you ship **editing to a local client**, you will see this in your editing buffer:

```
----------------------------------------------------------------------
@program #2108:add
.
----------------------------------------------------------------------
```

[19] Something you could have done is to assign an inital string to the property, like in '@property holder.holding "Anna Maria"' But don't want to do that either in this case!

DON'T delete the first and the last line and insert all programming lines in between those! To ship the text back to the MOO in the **rmoo client**, type `ctrl-c ctrl-s`. In other MOO clients, the command may be different. Note that you can reuse the same buffer as many times as you like. In other words, once you are editing a verb you can make as many modifications as you like and ship them back to the MOO each time.

(2) Alternatively, **if you use the in-MOO Verb editor** (discouraged but sometimes you have to) you have to learn how to use it (type 'look' and 'help' and you will see something like this:

```
----------------------------------------------------------------------
Verb Editor

Do a 'look' to get the list of commands, or 'help' for assistance.

Now editing #111:test (any any any).
----------------------------------------------------------------------
```

Now back to programming. The "add" verb needs to do 2 basic things: (1) add the "information bit" to the "holding" property and (2) inform you that it did so. To do that we need to program the following lines:

```
  --------------------------------------------------------------------
  this.holding = {@this.holding, dobjstr};
  player:tell("Inserted ''", dobjstr, "'' to ", this.name);
  --------------------------------------------------------------------
```

If you are using a MOO client that does local editing (tkmoo lite or emacs for instance, you should produce something like this in your editing window (the object number will be different of course):

```
--------------------------------------------------------------------
@program #2108:add
  this.holding = {@this.holding, dobjstr};
  player:tell("Inserted ''", dobjstr, "'' to ", this.name);
.
--------------------------------------------------------------------
```

Those two lines already contain several features of the MOO programming language. If the MOO complains about bad syntax, like in the following example, you mistyped something:

```
Line 2:  parse error
1 error(s).
Verb not programmed.
```

Be sure to type in those 2 lines as above. Note that statements in the MOO language need to separated by ";" so don't forget them! Errors are usually due to:

1. Missing ";"
2. Mispelling of builtin functions or verbs
3. Unmatched parenthesis or quotes

You can now try out your new verb. Use it as in the specification above (e.g. type 'add “anything you like” to holder' to the MOO). If you want to be sure that it worked, examine the object “holder”. E.g. type:

```
@d holder.
  - or -
;#<object>.holding
```

The “;” lets you evaluate simple expressions directly in the MOO without really programming a verb. Now let's look at the first line (reproduced below) in the “add” verb you started programming.

```
this.holding = {@this.holding, dobjstr};
```

1. It assigns a value to the slot “holding” on the object on which the verb runs (the object named “holder” in our case). Assignments are done with a “=” assignment statement. So permanent assignment statements are done in the following way:

   ```
   <object>.<property> = <something...>

   Examples:

   Player.name = "Animal" ;
   this.holding = {"not very much", "a lot more''} ;
   #237.xxx = 123+7-a ;
   ```

   The value of the expression to the **right** of the “=” sign is set as value of the property addressed to the left. To access the “holding” property on the “holder” object, we can use a construction like `this.holding`. “this” is a built-in variable that represents the object on which the verb runs. Those built-in variables allow you to write code that can be used by a whole class of objects.

2. On the right hand side of the “=” we find a list construction expression: `{@this.holding, dobjstr}` which does the following thing:

   - the `{ ...}` builds a list out of the comma-separated elements.
   - In this context, the “@” is a splicing operator that instructs the server to take not the list itself but its elements.
   - I.e. what we do here is telling the server to build a list from the elements which are already set to the property in the list (`@this.holding`) and to add the new element “dobjstr”.
   - “dojbstr” is another built-in variable that a verb can access and it corresponds to the “direct object representation” the parser found when a command was entered. In other words, if a user will type `'add "rdv demain avec Chris" to holder'`, the builtin variable `dbjstr` will have the value `"rdv demain avec Chris"` and the MOO will hand this value to the verb you are programming now. See section 7.5.2 on page 112 if you want to know more about those right now. Else go on!

If you don't understand the concept of splicing consider the following cases. In the first case we use an `@a` and the list created is a flat list. In the second case we don't use the `@` and we produce a list within a list, something we do not want here.

```
>;;a={"a","b","c"}; a={@a,"d"}; return(a)
=> {"a", "b", "c", "d"}

>;;a={"a","b","c"}; a={a,"d"}; return(a)
=> {{"a", "b", "c"}, "d"}
```

Note that we did not program a verb here but we used the **evaluator functionality** of the moo, which allows you to test simple things without writing a procedure. Please read section 11.5.1 on page 147 if you want to know more about playing around with the evaluator. But now, let's examine line 2:

```
player:tell("Inserted ''", dobjstr, "'' to ", this.name);
```

It runs the "tell" verb that (this time) is attached to the player, i.e. the person who typed the command. This line shows how to call other verbs from within a verb you are programming. The general syntax for verb calls is the following:

```
<verb>(<argument 1>, <argument 2>, <argument 3>, ...);
          e.g.
player:tell("Inserted ", dobjstr, "  to ", this.name);
```

i.e. arguments are given in parenthesis, separated by commas. [20] ":tell" accepts an arbitrary number of arguments, you can either use values or literals (e.g. strings in our case).

Be sure to put commas between EACH argument and watch out for quotes (") to be displayed to the player! Since " are used to delimit a string, you can't just have some " inside the string. Therefore, either use some other character to put quotes around words (e.g. '' (2 simple quotes) or better use **escaped** quotes: \". The following expressions are conceptually equivalent:

```
              "escaping" the quotes:

 player:tell("Inserted \"", dobjstr, "\" to ", this.name);

              or using any kind other character(s):

 player:tell("Inserted ''", dobjstr, "'' to ", this.name);
 player:tell("Inserted `", dobjstr, "' to ", this.name);
 player:tell("Inserted <<", dobjstr, ">> to ", this.name);
```

It is left as an exercise to the reader to figure out what each argument to the ":tell" verb will print. Play around with the MOO evaluator if you want, however the dobjstr variable will be empty, but something like this would work:

```
;player:tell("this is my ", player.name)
```

[20] Note that arguments are not "typed" as in PASCAL in the MOO language. It is up to you to know how many arguments you have to use and what type the verb in question can handle!

6.3.4 The second verb "dump":

Now we want you to be able to show what information is stored in the object. So we want to be able to say something like: 'show 1 on holder' to display the first element or 'show all on holder' to display all elements to everybody in the same room.

Before we do that, let's program a simple "dump" verb that will dump the contents of the object to yourself only. If we want a verb that works like 'dump holder' we first must create a simple verb like this:

```
>@verb holder:dump this none none
```

Now you have two verb specifications as you can see with the @d command (be sure to type the ":" after "holder"):

```
>@d holder:
holder (#2108) [ readable ]
  Child of generic thing (#5).
  Location island (#1429).
 #2108:add      MooBoy (#1324)       r d    any at/to this
 #2108:dump     MooBoy (#1324)       r d    this none none
```

Note, that if you don't know the `@d` command yet you can go and read section 11.2.1 on page 142. Below is the code for the "dump" verb. Enter it now and read the explanation of what it does below:

```
----------------------------------------------------------------------
for item in (this.holding)
  player:tell(item);
endfor;
----------------------------------------------------------------------
```

Here we use an iteration construct, i.e. the "for looping statement". (See also Iteration Section[21] in the MOO Programming Manuel[22] if you want to know more about iteration).

In the above case the "for" statement does the following thing:

1. It evaluates the expression `(this.holding)` which must return a list. Luckily we know that the "holding" property is a list.
2. Then it cycles **once for each item** through every statement between the "for ...." and the "endfor;" keywords. I.e. in our case it will call the "player:tell" verb for each item in the "holding" list and print it out.

The "for statement" is a more complex one then the "assignment" statement we have seen before. It can contain other statements. You did use some very important programming concept here: the iteration, i.e. doing something several times over. In our case we applied the `player:tell` verb several times, i.e. once for each item in the in the "holding" property.

Now let's try out what we have done so far: Add a few items with the "add" verb like in the transcript below and then type "dump holder" to see what happens.

[21] http://tecfa.unige.ch/guides/MOO/ProgMan/ProgrammersManual_32.html#SEC32
[22] http://tecfa.unige.ch/guides/MOO/ProgMan/ProgrammersManual_toc.html

```
>add "item 1" to holder
Inserted ''item 1'' to holder
>add "item 2" to holder
Inserted ''item 2'' to holder
>dump holder
item 1
item 2
```

Now let's improve this program a bit. It would be nice to add numbers in front of each item to be displayed for various reasons. The following code will do that. An explanation of what it does is below and might read it first. To edit this verb again, just add the necessary lines in your editing buffer or edit the "dump" verb again if you use an in-MOO editor.

```
---------------------------------------------------------------------
n=1;
for item in (this.holding)
  player:tell(n, ": ", item);
  n=n+1;
endfor;
---------------------------------------------------------------------
```

Now try out again the "dump verb" you just programmed. If you think that you don't have enough information in your holder, just add a few items with the "add" verb, e.g. type: `'add ''new banana'' to holder'`.

In this variant of the "dump" verb, we used a classic programming construct: We added and displayed a counter. We also made use of a temporary variable "n" that only lives as long as the verb executes.

This is how counters work:

1. a local variable acts as counter, i.e. it is holding an integer number that will be increased after each loop. In our case we call it "n".
2. each time we cycle through a loop, the counter is increased by 1. In our example we use the statement `n = n+1;`
3. Last, but first the counter has to be initialized, typically to a value of either 1 on 0. In our example the `n = 1;` statement does this. Note that this statement takes place before the "for" statement.

6.3.5 The show verb

Now let's program a verb that allows you to display an item to the people in the room or everything. So we want a verb that does something like:

```
show 3 on holder
```

This would display item 3 in the list if it existed. If it doesn't it would warn the player that such an item does not exist.

```
show all on holder
```

should display all information in the holder. Here is the verb and you can grab the verb from here[23]. Now let's examine the code. Note that to list a verb in the MOO you can use the @list command as below. Please note, that the line numbers in front of each lines are added by the '@list' command for your convenience. So don't ever type them when you program the verb yourself! It is useful to display line numbers when you quickly want to list a verb in order to see where it broke (grin).

```
>@l holder:show
#2108:"show"   any (on top of/on/onto/upon) this
 1:"This verb will print out either all objects or a selected item on the holding";
 2:"property to the room. Tests are made to insure that the direct object string";
 3:"represents a number AND an number not bigger than the length of the list";
 4:  n_items = length(this.holding);
 5:  " --- showing all the items ---- ";
 6:  if (dobjstr == "all")
 7:    player.location:announce_all(player.name, " shows all the items on ",
       this.name);
 8:    for k in [1..n_items]
 9:      player.location:announce_all(k, ": ", this.holding[k]);
10:    endfor
11:  else
12:    " ---- testing bad dobjstr's ----";
13:    NUM = tonum(dobjstr);
14:    if (NUM == 0)
15:      player:tell("The correct syntax for the show command is:
         'show all on <object>' or show <number> on <object>");
16:    elseif (NUM > n_items)
17:      player:tell("Sorry this list has less items then ", NUM);
18:    else
19:      " ---- displaying a single element ----";
20:      player.location:announce_all(player.name, " shows item ", NUM,
         " on ", this.name, ": ", this.holding[NUM]);
21:    endif
22:  endif
--
.
```

Now, please program this verb by entering the code or by copying it:

1. First, create the verb argument specification like this. In case your are working at **TECFAMOO skip this step** and go directly to the next step, i.e. use the @copy command (see next step) without defining the verb first.

    ```
    @verb holder:show any on this
    ```

23 http://tecfa.unige.ch/moo/book2/code/code-show.moo

2. Then program the verb in the same way as you programmed your other 2 verbs, i.e. type: '@edit holder:show'. Now instead of manually typing the code, you can grab it over the Net (see above) and paste it into the editing buffer (or whatever technique for uploading code you prefer). At **TECFAMOO** you can simply `'@copy #2108:show to <your object>'`. This will copy the verb from and object on which it allready exists to your object.

Then, play around with the verb trying out various variations to convince yourself that it works. This verb is not perfect and we suggest that make some modifications later. The main goal of this part of the tutorial is to teach you how a basic algorithm works and how to understand somewhat longer code. In case you don't understand in detail how this code works, there are 3 solutions:

1. You try to figure out by "hand" what the code does, i.e. you try to imagine step by step what will happen when this procedure runs.
2. You insert a lot of "print" statements in the code that will print out variable values. To do that, you can insert either something like:
   - player:tell("variable k =" , k);
     This solution is not great since lists will be printed as "list"
   - player:notify("variable k =" , k);
   - player:report("variable k =" , k);
     works only in certain MOOs.
3. The best solution is to run TECFA's (Boris Borcic's) Code animator in the Xemacs client. It will in the editor buffer replace variables by values in the code in step by step fashion, so you can really see what the code does.

Now let's examine new features of the MOO language we use in this verb.

1. You can insert comments in your code that will be remembered by the server. Simply write a statement enclosed by quotes, e.g.

   ```
   " ---- displaying a single element ----";
   ```

2. In this verb we use another variant of the "for" loop statement. In the "dump" verb we used a "do list" construction, whereas here we use a "do times" construction, i.e. we don't tell the loop to run once over each item of a list, but we tell it to run N times (from number n1 to number n2)

   ```
   for <variable> in [<beginning>..<end>]
   ```

   Here is an example:

   ```
   for k in [2..10]
   ```

   will do 2 things. (1) It will loop `<end> - \verb<beginning>` times , e.g 10-2 = 8 times in the above example. (2) It will assign the "index" of the current loop to the counter `<variable>`. I.e. the loop will assign k=2 in the first loop and finish with k=10. Note that in the "show" verb we use the iteration index "k" to access the nth element in the list hold by the "this.holding" property.

3. Elements of lists can be accessed with the following syntax (see the section on Indexing into Lists and Strings[24] in the MOO Programmers Manuel for more details):

   ```
   <list>[<element number>]
   ```

   ```
   Examples:
      {0,1,2,3,4}[2]    ==> will return 1
      this.holding[3]   ==> will return the 3rd element
   ```

   Note that accessing lists out of range (e.g. element 5 in a list of 4 elements) will produce an error ! The built-in "length()" will tell you the length of the list and you can test list access like we did in the ":show" verb:

   ```
    4:  n_items = length(this.holding);
    ......
   16:    elseif (NUM > n_items)
   17:      player:tell("Sorry this list has less items then ", NUM);
   ```

4. The most important new item of the MOO programming are conditional statements (IFs). See the section on Statements for Testing Conditions[25] in the MOO programmers Manual. Here we just reproduce an abstract pseudo code definition of what happens in the "show verb".

   ```
    6:  if (``the direct object string is EQUAL to "all")
        ``do print out all statements'';
   11:  else
   14:    if (``the direct object string is not a number, e.g. 0!'')
          ``then complain to the user'';
   16:    elseif (NUM > n_items)
          ``else if we were able to create a number BUT the number is
           too big complain too'';
   18:    else
          ``print a single element;
   21:    endif
   22:  endif
   ```

   So IFs can be nested and alternative conditions "else" or "else if" exist. Note that the (conditional) and the keywords "else" and "elseif" belong to the same statement, so no ";" is needed for them! Here is the same same structure as in the "show verb" even more simplified.

   ```
   if (condition1)
     ``do something'';
     ``do something more complex'';
   ```

[24] http://tecfa.unige.ch/guides/MOO/ProgMan/ProgrammersManual_17.html#SEC17
[25] http://tecfa.unige.ch/guides/MOO/ProgMan/ProgrammersManual_31.html#SEC31

```
else
  if (condition2)
     ``do something else'';
  elseif (condition*)
     ``do something else'';
  else
     ``do something else again''
  endif
endif
```

5. Conditional expressions are used for example in IF statements and they return a truth value, i.e. the expression on line 6:

   ```
   (dobjstr == "all")
   ```

   is either 1 (= true) or 0 (=false). Please remember that equality tests are done with the "==" operator and NOT with a simple "=" which is an assignment.

6. The tonum() built-in function in line 13 will translate a string to a number. If conversion is not possible, the number is 0. Try it out by typing ';tonum("bla") to the MOO client. The length() function in line 4 returns the length of a list. As we explained in section 6.2.4 functions always return a value. You will learn later how to build your own function verbs. All built-in functions are explained in the MOO manual or in the MOO (type 'help functions').

7. An other built-in function allows displaying messages to the people in the same room. Each room (normally) has a "announce_all" verb that works in a similar way as player:tell. The latter only displays something to the player who typed the command. Each player has a location property containing the room object the player is currently in. So all we need to run this verb is to use the following kind of statement:

   ```
   9:      player.location:announce_all(k, ": ", this.holding[k]);
   ```

That's it ! Again: try to study this verb until you are sure how it works. As we said you can insert statements that print out intermediary variables or whatever you want to you. I.e. if you want to be sure that the counter works right in the for loop, insert this into the loop after line 8 or 9:

```
player:tell("K= ", k);
```

6.3.6 Cleaning up

To make this holder object more useful we need at least the 3 following functionalities: (1) A verb for removing elements, (2) a verb for clearing all messages and (3) a help message attached to the object, so that 'help holder' would work.

A simple "clear" verb is really simple as you can see below.

```
@verb holder:clear this

>@l holder:clear
#2108:"clear"   this none none
 1:  this.holding = {};
 2:  player:tell("You erased all information bits from ", this.name);
--
```

Of course this verb is highly dangerous. People might erase all their information by mistake. So what we should do actually is asking confirmation from the user, like in the example below. You can add that verb now. Note that we won't tell you any more "edit the verb please and such", you should know that by now from the description given below.:

```
>@l holder:clear
#2108:"clear"   this none none
 1:  answer = $command_utils:yes_or_no
     ("Do you really want to erase all information bits?");
 2:  if (answer)
 3:    this.holding = {};
 4:    player:tell("You erased all information bits from ", this.name);
 5:  else
 6:    player:tell("OK, information on", this.name,
       " was too important for erasing");
 7:  endif
```

The only new things here is are in the first 2 lines: `$command_utils:yes_or_no` is a Lambda Core utility function. Those functions are live savers, so please look at them as soon as your really start programming. Utility functions are verbs attached to various $objects. Type 'help utils' for a list of the different packages. Note that you NEED command utilities for interacting with users.

The `:yes_or_no verb` returns 1 if the persons types "yes" and 0 for anything else (including "no"). The conditional (answer) is true if answer is set to 1, otherwise false and the else clause will be executed.

Finally we will show you an erase verb that does not ask confirmation. You should add this functionality as an exercice if you wish to have it.

```
@verb holder:erase any from this

>@l holder:erase
#2108:"erase"   any (out of/from inside/from) this
 1:  NUM = tonum(dobjstr);
 2:  if (NUM == 0)
 3:    player:tell("The correct syntax for the show command is:
       'erase <item> from <object>'");
 4:  elseif (NUM > length(this.holding))
 5:    player:tell("Sorry this list has less items then ", NUM);
 6:  else
 7:    player:tell("You erased element ", NUM, ": \"", this.holding[NUM],
```

```
      "\" from ", this.name);
      this.holding = listdelete(this.holding, NUM);
    endif
--
```

Additional exercice: If you wish you could also ask the user for confirmation before you erase an item.

Now, sometimes people don't want to type in commands like 'show all on holder' and then type 'erase 3 from holder', but rather have all elements displayed and then have the program ask which elements you want to erase. Even better we could write a verb which also would let you add elements. All this in "menu style". We will address this issue in an other tutorial.

6.3.7 Adding help

Help messages on objects are attached to the "`.help_msg`" property. Type '`@notedit holder.help_msg`' and produce a more serious help than the one below (emacs notediting buffer shown):

```
@set-note-text #2108.help_msg
This is an exercice left for the reader!
Type @examine in the meantime....
.
```

6.3.8 What is missing?

We can think of several things, e.g. it is not a generic object, meaning that other persons (users of the MOO) can't make a child of it (have a copy for themselves), see the generic object tutorial (section 6.6 on page 99) for learning how to do that.

Another problem is security, as you can see anybody will be able to write something onto your "holder" or worse, erase it. This is something else you will have to learn if you feel getting into serious MOO programming.

6.3.9 La morale de l'histoire

The little object we programmed respects quite well object-oriented programming style (well normally one does not program individual objectcs but rather generic ones, but would not change programming style). We respected:

Modularity: Modularity: Verbs only operate on slots of the object to which they belong.

Why is that important ? Let's assume a player wanted to know how much information he remembered and deleted in a given interval of time.

Now we could:

- put a "info_counter" property on the player (on yourself you are allowed to do that),
- and then for example count changes like this in the holder's "add" and "remove" verbs:

```
player.info_counter = player.info_counter + 1;
```

This is not a very flexible way of doing things. It is MUCH better to write a verb on the player called something like "`update_info_counting`" that does the job. This way we could customize information counting for different players, or program a turn-on/off function. Don't try to program this for the moment, since it will only work on your own character. Rather go to the Generic Objects and Permission tutorial.

Going further The code you wrote is hopefully a good example on how to use the MOO programming language, but it is a very example on how you should develop software. Things that are missing are:

1. This code can only be used for you, but by nobody else! In the MOO you have to write code that can be used by others.
2. Abstraction of Code Segements.
3. Generalization of function elements
4. Specialized "around" verbs.

[ok I have to say a bit more here]

Once you are done with this tutorial, you can try to add functionalities to this objects, or have it behave in different ways. As an example, look at Thora's flower (`#2783 @ TecfaMOO`) and figure out the differences.

6.4 Social verb tutorial, level 0

This section will teach you how to write simple "social" verbs, i.e. verbs that just send messages around and do not modify the contents of any object. Ususally that kind of verb is made available in social features, but before writing and propagating those, it is better to write a few just for yourself. In order to do the "Webby Tutorial 0" (see section 6.5) and the Generic Objects and Permission Tutorial (see section 6.6) you don't need this section. So not much "tutoring" is done here and we rely on you for picking up the challenge! Feel free to pick it up later (i.e. once you know how to make generic objects and feature objects).

The "grumble" verb Get the code from here[26] if you want to play around with this verb. Then write your own social verb !

This verb shows a few more MOO programming tricks, all related to sending messages to users:

1. This time we use a "any to any" verb specification. The reason is the following one. We want to say something like:

   ```
   grumble ''now this is really too much'' to somebody
   ```

[26] http://tecfa.unige.ch/moo/book2/code/code-grumble.moo

The first "any" specifier means that we can type in anything (e.g. a string) and the second "any" specifier means that we do not operate this verb on a specific object; "any" in this means a random user name.

2. Since the MOO parser does not find the addressee for us over distance (players in other rooms) we have to use a Lambda core utility function:

   ```
    9:    addressee = $string_utils:match_player(iobjstr);
   ```

   The "iobj" builtin variable will match IF the person or an object is in the same room, if not we set this variable to the player object we found (if any is found).

3. Contrary to the "holder" tutorial (see section 6.3) we give differentiated feedback to the player who types the command, to players in the same room and so forth, e.g.

   ```
   32:    player.location:announce_all_but({addressee, player},
          player.name, " grumbles", mess);
   ```

4. In order to page a person in a other room, we use the following verb:

   ```
   18:    addressee:receive_page("You hear " + player.name +
          " grumbling at you over distance: ", "    \"" + dobjstr + "!\"");
   ```

Here is the code. Please note that some lines have been broken for display here.

```
@verb me:grumble any to any

----------------------------------------------------------------------
>@l ~mooboy:grumble
@program ~MooBoy:grumble   any at any
 1:  "The grumble verb allows you to grumble at persons + objects in the same room";
 2:  "as well as to persons in other rooms. Players in the same room will see";
 3:  "your message, players in the distant room will just notice a grumbling";
 4:  "The only thing that sucks is that this is a any to any verb";
 5:  "Try 'grumble this is a test to xxx' to see what I mean";
 6:  "";
 7:  " --- test if the object or person is in the same room";
 8:  if (iobj == #-3)
 9:    addressee = $string_utils:match_player(iobjstr);
10:    " ------ if not, is it a player ?";
11:    if ($command_utils:player_match_failed(addressee, iobjstr))
12:      player:tell("This player does not exist nor is there
         any object of that name in this room");
13:    else
14:      " ------ it is a player and we can try to page it";
15:      if (addressee:is_listening())
16:        player:tell("You grumble-page to ", addressee.name,
```

```
      " [", addressee.location.name, "]: \"", dobjstr, "\"");
      addressee:receive_page("You hear " + player.name +
      " grumbling at you over distance: ", "    \"" + dobjstr + "!\"");
      addressee.location:announce_all_but({addressee},
      "You vaguely hear that ", player.name, " grumbles something at ",
       addressee.name);
    else
      player:tell($string_utils:pronoun_sub("No need to grumble,
       %s is not connected right now", addressee));
    endif
  endif
else
  "------ the thing is in the same room, grumbling face to face";
  addressee = iobj;
  mess = " \"" + dobjstr + "\" " + prepstr + " " + addressee.name;
  if (is_player(addressee))
    player:tell("You grumble", mess);
  else
    player:tell("The ", addressee.name,
    " is not overly impressed by your grumbling.");
  endif
  player.location:announce_all_but({addressee, player},
  player.name, " grumbles", mess);
  addressee:tell(player.name, " grumbles at you, ", "\"", dobjstr, "!\"");
  addressee:tell(" ... maybe you should really start worrying...");
endif
/* Last edited on Tuesday, June 25, 1996 at 11:40.38 am by MooBoy (#1324). */
----------------------------------------------------------------------
```

Of course, this verb is not perfect yet. One thing missing might be permissions. right now this verb can be used by an other person that is in the same rooms as shown in the following examples:

An other person (e.g. Daniel) that is in the same room as Mooboy could type:

```
>grumble "something" to MooBoy
You grumble "something" to MooBoy
```

and MooBoy would see:

```
Daniel grumbles at you, "something!"
... maybe you should really start worrying...
```

This works because the MOO parser will find the “grumble” verb on MOOBoy who is in the same room. Grumbling over distance would not work as shown in the following example. The reason is that no matching verb has been found either on the player typing the command, nor the room, nor any object that has been matched to the command typed.

```
>grumble "watch out" to Thora
I don't understand that.
```

We will address this and other security problems in an other tutorail [DOIT!], but you can look at sections 8.2 (general) or 9.4 (for specific information concerning **TECFAMOO** .

Another problem that could be easily solved is that you can grumble at persons in the same room that are disconnected. No harm done, but one could filter that situation too.

The "grab" verb Now let's look at a similar social verb, it is a simpler form of the previous verb, but has a permission check and some randomness built-in:

1. In line 11 we set the variable messages to a list of messages, something like `messages={``a'', ``b'', ``c'',}`
2. In line 12 we select a random element of this list that will we displayed. `message = messages[random(length(messages))];` uses the builtin random function to select an element of the list. Note that `random(length(<list>)` simply means select a random number between 1 and the length of the list (messages in our case).
3. In the beginning of the verb (lines 6-9) we check if the caller of the verb is the same object as the object to which the verb belongs (you in this case). If this is not the case we print out a message to the caller, i.e. the person who typed the command.

```
>@l me:grab
@program ~MooBoy:grab   any none none
 1:  "The grab verb allows you to ``grab'' persons + objects in the same room";
 2:  "as well as persons in other rooms";
 3:  "All it really does is just displaying a random message to the other person";
 4:  "";
 5:  " ---- permission: only the owner can execute this verb";
 6:  if (caller != this)
 7:    player:tell("Sorry, you don't have permission to use this verb");
 8:    return;
 9:  endif
10:  " ---- list of messages and random generation of one --- ";
11:  messages = {".... maybe you really should start worrying a bit !",
 "You feel that this might be the start of something really dangerous",
 "You sense his virtual eyes on you!",
 ".... maybe you should disconnect",
 "Did you say good bye to your friends ?,
 you might just want to do that before it's too late!"};
12:  message = messages[random(length(messages))];
13:  " --- test if the object or person is in the same room";
14:  if (dobj == #-3)
15:    addressee = $string_utils:match_player(dobjstr);
16:    " ------ if not, is it a player ?";
17:    if ($command_utils:player_match_failed(addressee, dobjstr))
18:      player:tell("This player does not exist nor is there
         any object of that name in this room");
```

```
    else
      " ------ it is a player and we can try to page it";
      if (addressee:is_listening())
        player:tell("You grab ", addressee.name,
         "[at ", addressee.location.name, "] ", "over distance and ",
         addressee.ps, " sees: \"", message, "\".");
        addressee:receive_page("You feel " + player.name
        + " grabbing you over distance!", message);
      else
        player:tell($string_utils:pronoun_sub("No need to grab,
        %s is not connected right now", addressee));
      endif
    endif
  else
    "------ the thing is in the same room, grumbling face to face";
    if (is_player(dobj))
      addressee = dobj;
      player:tell("You grab ", dobjstr, " and ", addressee.ps,
      " sees: \"", message, "\".");
    else
      player:tell("The ", addressee.name,
      " is not overly impressed by your grabbing.");
    endif
    addressee:tell(player.name, " grabs you, ", dobjstr, "!", message);
  endif
  /* Last edited on Tuesday, June 25, 1996 at 2:04.43 pm by MooBoy (#1324). */
--
```

6.5 [TECFAMOO] "Webby" Tutorial, Level 0

Note that this tutorial will only work at **TECFA**MOO , MooTiny or other MOOTiny compatible MOOs! Even, you might find things that don't work exactly the same way!

Let's webbify the object we built in section 6.3. To do so (1) drop your holder on the floor and log into the web as the same character. You will land at the same location you are in the MOO client. Then, click on the object "holder".

Documentation on the www layer:

1. Type 'help www-index' in the MOO (partly BROKEN)

2. For people interested in the mecanism behind the WWW layer see chapter 10, section 10.1. But applications programmers don't need to read this.

3. See section 10.3

6.5.1 Adding a webby description

The first (optional) task is to add something web specific to the description of the object. To do that you simple add HTML tags to the ".htext" property. This slot can be edited with the note editor like a description. So type

```
@notedit holder.htext
```

You can insert any valid html, BUT you don't need to put `<html>` and `<body>` tags around your text, since our WOO application layer will insert this text automatically at some place in the body. Every object on **TECFAMOO** (and MOOtiny oriented WOOs) has an ".htext" property you can edit.

6.5.2 Adding webby verbs

"Webby verbs" are verbs displayed as buttons at the buttom of the page and that allow you to manipulate the object. If you wonder what a "webby verb" is click on the help button on the "holder" page. This verb will display the standard help message plus all the comment strings it finds on top of the objects verbs.

"Webby" Verbs are by convention all verbs whose name terminates by "_web". E.g. the inherited help verb on each object is `#1:help_web`. You should have a look at it sometimes. Why do webby verbs work? Have a glance at sections 10.3.1 and 10.3.2!

A simple web verb: Let's program a simple verb that will show the contents of the holder. In order to do that we have to introduce a new type of verbs, i.e. verbs that are called by other verbs. Such verbs (i.e. ones not typed in by the user as commands) are by convention **"this none this"** verbs. Therefore we create our verb like this:

```
>@verb holder:show_web this none this
```

Now please display the verb argument specifiers and the permissions by typing: `'@d holder:show_web'`. In case you forgot the "this non this" you got to do the following thing (use 'help ...' if you want to know more right now):

```
@args holder:show_web this non this
@chmod holder:show_web +x
```

Programming simple "webby" verbs without buttons is quite simple as you can see in the example below. The only really new thing here is that we use verbs as functions. Functions are cool for letting application programmers use complex software packages such as the Web interface. You may want to read section 6.2.4 to see why. In section 6.3 we used verbs as procedures, i.e. algorithms that that work on data-structures but do not return values for other use. Note, however that builtin-in functions exactly did that. So what we ask from now is to write function verbs according to some specifications, i.e. they must return a certain type of value and they will accept certain arguments as input. You must follow those specifications, else nothing will work.

The show verb has the following specifications:

1. It does not take any argument (note that "_web" verbs may take, but here we decided to program a simple button on which you can click)

2. It MUST return a FLAT list of strings, each representing some HTML code.

Now, copy the verb onto your holder object and check if it works over the www interface.

```
>@l holder:show_web
#2108:"show_web"   this none this
 1:  "This link will show you the contents of holder";
 2:  header = "Show contents of" + this:title();
 3:  item_list = {};
 4:  for item in (this.holding)
 5:    item_list = {@item_list, "<li>", item};
 6:  endfor
 7:  return {@$www_utils:do_html_header(header), tostr("<h1 align=\"center\">",
     header, "</h1>"), "<ol>", @item_list, "</ol>", @$www_utils:do_html_footer()};
--
```

Now let's examine line 7. This line returns a list to a verb that has called this function. The list is the HTML code that will be displayed in the body of the page which will be generated in reply to activation of this webby verb. This list must be a FLAT list of strings representing HTML code.

Note that the top of all html pages should stay the same by convention (well you might decide otherwise). Now let's see in detail what kind of list the return statement returns:

building the header @y\$www_utils:do_html_header(header) will build:

- the header of the page, e.g. something like <html> <head> <title> .... </title> </head> <body>
- the menu that appears normally on top of each woo page.
- the main heading (see below)

building the main heading tostr("<h1 align=\"center\">", header, "</h1>") will build the main title (h1) of the page

generating the list of information bits "
", @item_list, "
" will build an <ol> ... type of list, and insert in the middle the elements of the item_list we built above.

adding the footer @\$www_utils:do_html_footer()} will add a footer to the html page, e.g. things like <\body> <\html>.

A Web verb with a text box "Webby" verbs with arguments are more trickier. Arguments to displayed in the web are specified in comment lines at the beginning of the verb. (See section

```
>@l holder:add_web
#2108:"add_web"   this none this
 1:  "text to add: ?txt70?.";
 2:  player:report(args);
 3:  header = "Add element to " + this:title();
 4:  text_to_add = args[1][1];
 5:  player:report(text_to_add);
 6:  this.holding = {@this.holding, text_to_add};
 7:  return {@$www_utils:do_html_header(header), tostr("<h1 align=\"center\">", header
--
.
```

6.5.3 Customizing the :htext verb

The verb that diplays most of the object on the web is ":htext". If you wish you can edit the verb to display information it normally would not. This verb is not too complicated, but you have to watch out what you do: It has to return some correct html ! Also to insure that people won't get lost, it is a good idea to keep the standard header of the html page.

The most important core objects have :htext verbs defined (see section 10.3.2 for details). So our holder verb inherits the :htext verb from $thing which in turn inherits it from #1.

Please note that the verb below may change now and then ! Also remember that we break up lines here to make them fit on a page (especially the paper version of this manual)

```
>@l #1:htext
#1:"htext"   this none this
 1:  "Returns the object:title as both the html title and as an <h1>
      header at the top of the web page.";
 2:  "Followed by the objects description, and checks to see if text
      is preformatted.";
 3:  "Generates a url for the object's location, adds in anything from
      the htext property, followed by any web_verbs, and ending with
      a standard footer.";
 4:  set_task_perms(caller_perms());
 5:  title = this:title();
 6:  desc = typeof(this.description) == LIST ? this.description |
     {this.description};
 7:  if (this.pre)
 8:    desc = {"<pre>", @desc, "</pre>-----<br>"};
 9:  endif
10:  loc = valid(this.location) ? tostr("<h3>Location:</h3> ",
     $www_utils:genref_obj(this.location)) | "";
11:  htext = typeof(this.htext) == LIST ? this.htext | {this.htext};
12:  verbs = this:list_web_verbs();
13:  "this.webhits = this.webhits + 1;";
14:  return {@$www_utils:do_html_header(title),
      tostr("<h1 align=\"center\">", title, "</h1>"), "<p>",
```

```
      @desc, "<p>", loc, "<p>", @htext, "<p>", @verbs,
     @$www_utils:do_html_footer()};
15:  "Last edited by Irradiate on Fri Aug 25 14:47:32 1995 MET DST";
--
```

You can copy this verb an tailer it for your needs if you wish. Be sure to delete the following line.

```
4:  set_task_perms(caller_perms());
```

Since :htext verbs on core objects run under wizard permission execution permissions are changed at run-time. But only wizards can do that.

Also, if you really plan to write your own :htext verb, consult the **$www_utils** box, or at least try to understand what the **$www_utils** verbs used in the above :htext verb do.

6.6 Permissions and Generic Objects Tutorial

The main goal of this tutorial is to teach you how to write generic objects that can be used by other persons. In order to do this tutorial, you must have passed trough the “holder” tutorial in section 6.3 or else adapt instruction to another object you own.

You have programmed an object and you think it might be useful to others (or least you want several instances for yourself), i.e. you could do something like:

```
@create <your object> named <a name>,[<alias name>]
          e.g.
@create #13456 named Black Holder, hold
```

There are 2 things to do:

1. You have to make this object fertile (so that people can create “kids” of it)
2. You have to insure that property permissions are done right.

Before we address that topic let’s first have a general look at permissions.

6.6.1 A short look at Permission

1. Every object, property, an verb has a set of permissions
2. If you own an object, it does not necessarily mean that you own all the properties and all the verbs. To convince yourself, look at an object, e.g. type ’ **@d me.**’, this will list the properties of your character.

The table below gives a short and dirty overview on existing permissions:
Let’s point out the highlights first and look into “f” and “c” permissions in the next section.

- General:

	Suggestions for each kind		
Permission	Objects	Properties	Verbs
r (read)	ok	ok	ok
w (write)	never	almost never	never
x (executable)	-	-	use for "this none this" verbs
f (fertile)	use for generics	-	-
d (debug)	-	-	mostly
c (create)	-	sometimes in generic objects	-

Table 6.1: A short summary on permissions

1. It is good custom to set read permissions on everything, because then people can look at your code, and decide whether it is safe to use it. "r" is default at **TECFAMOO** .
2. Write permissions, means that EVERYBODY could:
   - add components to your object (object=w)
   - reprogram a verb (verb=w)
   - change the contents of properties (property=w).

   Only sometimes it can be useful to set a property to "w", i.e. if want people to set a slot representing information or so.

- Verbs

1. "x" permissions are NEEDED for all verbs that are called by other verbs. If you create a "this none this" verb, x is added by default (but not if you change the argument specifiers of a verb with the '@arg' command! You can set command verbs to "x", if you wish to test them with the ';' (eval) or if you want to call them from other verbs.
2. "d" permissions on verbs are recommended because if the verb breaks on an error it will print out "feedback".

Inspecting and Manipulating Permissions: You can look at permissions by using the @display ('@d) command like in the following transcripts:

```
   >@d holder
   holder (#2108) [ readable fertile ]
     Child of generic thing (#5).
     Location island (#1429).
-------------------------------- finished -----------------------------
```

The object holder is readable and fertile

```
   >@d holder:show
    #2108:show      MooBoy (#1324)   r d    any on top of/on/onto/upon this
-------------------------------- finished -----------------------------
```

The :show verb is "r" and "d": readable and debug

```
>@d holder.holding
.holding          MooBoy (#1324)        r      {}
------------------------------ finished -----------------------------
```

The ".holding" property is just readable. Now compare that to your holder. If did not change anything since your worked last on your holding object, permissions should not be the same.

To change permissions, use the '@chmod' command (type 'help @chmod' in the MOO). E.g.

```
@chmod holder +f
```

will make it fertile.

```
@chmod holder +w
@chmod holder -w
```

will make it writable (don't do that) first and then lock it for writing again.

6.6.2 A generic "holder"

The first an easy step in creating a generic object is to change permissions to make it fertile. Do that now !

Now we have to mention a somewhat tricker issue: the "c" permissions. Here are the ground rules:

1. If you have a c permission set on a property, the owner of the kid will own this property, meaning they can set it to any value they wish.
2. If you remove the c permission on a property, then the property all kids of your generic will be you. This means that the owner can't change them.
3. The trouble is that all inherited verbs run with permissions on YOU. This means that those verbs can not change properties on kids. Yes and verbs are NOT copied onto kids.

Now let's look at the holder example. You now should have it fertile, if not, do that. Now TELL SOMEBODY ELSE to create a kid of holder (give the person the object number or bring him to the room where your holder sits on the floor.

```
@create #... named test
```

Now tell him to 'add ...' something, but it won't work as you can see in the transcript below

```
>add this to test
#2108:add (this == #2263), line 1:  Permission denied
(End of traceback)
```

Now your friend should recycle the object, i.e. kill it because it is useless, i.e. he should type:

```
@recycle test
```

Note that YOU could create kids that work, since the owner of the generic and the owner of the kid is yourself.

The solution to the problem is to **change permission of all properties whose values are set by a verb on the generic to "-c"**. Now type something like this:

```
@chmod holder.holding -c
```

Now your friends could create kids of your holder that work. The trouble now is that they can't change those properties by hand (as some persons might be tempted to do, i.e. for them, something like:

```
;#12347.holding = {}
    or
@set test.holding to {}
```

would not work. If you want to allow that, there are 2 solutions both of which are not great:

1. You can change permissions on holder-holding to +w. This way both your verbs work and the owner can change the ".holding" property. BUT, anybody could do that too.
2. You can tell people to copy the verbs from you. But this is a waste of MOO space (unless they create a new and better generic object layer for the holding.

Without doubt, there is still more to be said about permissions, but for simple projects, you should be all set.

Further reading on permissions:

- The section on Permissions (7.7)
- The section on security (8.2)

Adding security Another problem is security, as you can see anybody will be able to write something onto your "holder" or worse, erase it. This is something else you will have to learn if you feel getting into serious MOO programming.

... sorry this is not done yet.

6.7 MOO software, Level0

In this tutorial, we will introduce a few software engineering ideas such as "modularity" and "abstraction layers". We will produce something like the "holder" of the tutorial in section 6.3 but more powerful and more flexible.

[soon]

Chapter 7

Elements of the MOO language

7.1 Introduction to the MOO system

Text for this section is mostly stolen from the Introduction[1] to the MOO Programmer's Manual.

LambdaMOO is a network-accessible, multi-user, programmable, interactive system well-suited to the construction of text-based adventure games, conferencing systems, and other collaborative software.

Participants (users, characters) connect to LambdaMOO using Telnet or some other, more specialized, client program. (See section 1.4 on page 13). Upon connection, they are usually presented with a welcome message explaining how to either create a new character or connect to an existing one. Characters are the embodiment of players in the virtual reality that is LambdaMOO.

Having connected to a character, players then give one-line commands that are parsed and interpreted by LambdaMOO as appropriate. Such commands may cause changes in the virtual reality, such as the location of a character, or may simply report on the current state of that reality, such as the appearance of some object.

The job of interpreting those commands is shared between the two major components in the LambdaMOO system: the **server** and the **database**. The server is a program, written in C, that manages the network connections, maintains queues of commands and other tasks to be executed, controls all access to the database, and executes other programs written in the MOO programming language. The database contains representations of all the objects in the virtual reality, including the MOO programs that the server executes to give those objects their specific behaviors.

Almost every command is parsed by the server into a call on a MOO procedure, or verb, that actually does the work. Thus, programming in the MOO language is a central part of making non-trivial extensions to the database and thus, the virtual reality.

Therefore, one needs to learn three things:

1. The MOO Programming language itself.

2. Extensions to the language that are included in the database you are using: libraries for doing common and rather low-level things.

[1] http://tecfa.unige.ch/guides/MOO/ProgMan/ProgrammersManual_1.html

3. How to use a core data base: A MOO without core is totally useless (unless you are an expert MOO programmer with a lot of free time), besides the above utilities it includes about 70 to 100 generic objects that will allow you to start building a virtual world, such as:

   - Rooms and exits
   - Different user classes
   - The mail system
   - Editing tools

   Most MOOs are derived from Lambda Core, but documentation is not always easy to obtain off-MOO. The LambdaCore User's Manual[2] Programmer's Manual[3] are outdated, but still might be useful. In most cases it is best to use the "in-MOO" help which is more up-to-date than those manuals.

7.2 The MOO Programming Manual and Lambda-type databases

The most important source for MOO programmers is the MOO Programmer's Manual[4]. Now note a few things:

1. The MOO Programmer's Manuel is not suited for total beginners. Go through chapter 6 on page 71 at the same time. Let's quote from the Manual: "MOO, the programming language, is a relatively small and simple object-oriented language designed to be easy to learn for most non-programmers; most complex systems still require some significant programming ability to accomplish, however". If you do know how to program some C-like language and preferably some lisp-like language too all you need to worry about fast are things like command parsing (see section 7.4), permissions (see section 7.7) and so on. Regarding those elements, think of the MOO as an operating system of a virtual reality.

2. Even if

3. It does not contain add-ons (libraries) that are part of the data base. Most data bases are lambda-moo based, though many

Server and Database Basics:

1. Introduction:

2. The LambdaMOO Database: Has information about MOO Value Types, Objects in the MOO Database (Fundamental Object Attributes - Properties on Objects - Verbs on Objects)

3. The Built-in Command Parser is important reading since it will explain how the MOO parses commands entered by the user.

[2] http://tecfa.unige.ch/cgi-bin/info2www?(LambdaCoreManual.info)
[3] http://tecfa.unige.ch/cgi-bin/info2www?(LambdaProgMan.info)Top
[4] /guides/MOO/ProgMan/ProgrammersManual_toc.html

The MOO Programming Language: Introduction

1. MOO Language Expressions
   (a) Errors While Evaluating Expressions
   (b) Writing Values Directly in Verbs
   (c) Naming Values Within a Verb
   (d) Arithmetic Operators
   (e) Comparing Values
   (f) Values as True and False
   (g) Indexing into Lists and Strings
      i. Extracting an Element from a List or String
      ii. Replacing an Element of a List or String
      iii. Extracting a Subsequence of a List or String
      iv. Replacing a Subsequence of a List or String
   (h) Other Operations on Lists
   (i) Spreading List Elements Among Variables
   (j) Getting and Setting the Values of Properties
   (k) Calling Built-in Functions and Other Verbs
   (l) Catching Errors in Expressions
   (m) Parentheses and Operator Precedence
2. MOO Language Statements
   (a) Errors While Executing Statements
   (b) Simple Statements
   (c) Statements for Testing Conditions
   (d) Statements for Looping
   (e) Terminating One or All Iterations of a Loop
   (f) Returning a Value from a Verb
   (g) Handling Errors in Statements
   (h) Cleaning Up After Errors
   (i) Executing Statements at a Later Time
3. MOO Tasks
4. Built-in Functions
   (a) Object-Oriented Programming
   (b) Manipulating MOO Values

    i. General Operations Applicable to all Values: Read this for conditionals.
    ii. Operations on Numbers
    iii. Operations on Strings
    iv. Operations on Lists

  (c) Manipulating Objects

    i. Fundamental Operations on Objects
    ii. Object Movement
    iii. Operations on Properties
    iv. Operations on Verbs
    v. Operations on Player Objects

  (d) Operations on Network Connections

  (e) Operations Involving Times and Dates

  (f) MOO-Code Evaluation and Task Manipulation

  (g) Administrative Operations

Server Commands:

1. Server Commands and Database Assumptions

Builtin Functions and TecfaMOO utilities:

1. TecfaMOO utilities via our not so finished MOO/WWW Gateway. Our utils are Lambda-core based.

2. Function Index

7.3 Objects and properties

[more to come]

7.3.1 Properties

Creation "Properties" are named slots in an object that can hold any kind of MOO Value (see section 7.5.1).

To create a property, you can '@property #obj.¡name¿' like in the following examples:

```
@property dog.times 0
@property dog.favorites {"Cats","Students", "bones"}
@property dog.color "Dark white"
```

```
  @property & <object>.<prop-name>
  @property & <object>.<prop-name> <initial-value>

  Adds a new property named <prop-name> to the named object.
  The initial value is given by the second argument, if present; it
  defaults to 0.

  Normally, a property is created with permissions 'rc' and owned by
  whoever types the command.  However, you may also specify these
  explicitly

 @property <object>.<prop-name> <initial-value> <permissions>
 @property <object>.<prop-name> <initial-value> <permissions> <owner>

  Only wizards can create properties with owners other than themselves.
```

When you create an object, you always get always the parents properties by default.

Modification: @set #obj.property to <something>.

Note that you can also use the editor to change the contents of the property. If the object is a list, you can use notedit.

Current versions of the MOO server initialize properties as 'clear'. This means that if a particular object is not the object that first defines a prop, then the prop will return the value of it's parent, and its parent's parent, (etc), until you reach the object where the prop is first defined. This means that if you change the value of the prop on an object that defines the prop, the value will change for any subsequent kids that have not had that value set.

The builtin is_clear_property() checks to see if a prop is clear, while the builtin clear_property() will return the prop back to it's cleared state. The best way to understand this is probably to just try it out with a few little eval experiments. Make a test object and don't describe it (doesn't matter what it's parent is). then type something like the following:

```
;is_clear_property(#test, "description")
```

This should return 1

Give your test obj a desc, and try the same eval. This time it should return a 0.

You can then take your test object and use

```
;clear_property(#test,"description")
```

to reset it's desc back to it's undescribed state.

Removing properties

```
Syntax:  @rmproperty <object>.<prop-name>
```

Removes the named property from the named object. '@rmproperty' may be abbreviated as '@rmprop'.

7.4 Verbs and command parsing

See also: Chapter 2 in the MOO Programming Manuel[5].

When a command is entered by a user, the server tries to (1) parse the line received and (2) to find a verb that can execute it.

7.4.1 Command parsing

The MOO server will parse a command that is entered according to several principles. Here we will just look at the most important.

1. It will replace some standard abbreviations (“””, “:”, “;”) by corresponding verbs.
2. Next the command line is split into “items”, i.e. every string will become an item and every word (separated by spaces and other separators (??) NOT in an explicit string will become an item. [6]
3. Next the “sentence” made up by those items is broken into either verb, direct object, preposition, indirect object or verb, direct object
   (a) The first item in the list is taken as the name of a verb
   (b) Next, the server looks for a proposition (one out of the predefined set). Items between the verb and before the preposition is considered to represent the direct object and all items after the preposition are considered to represent the indirect object.
   (c) If there was no preposition, everything that follows the first item (the verb) is considered to represent a direct object.
4. Next, the server will try to find MOO objects that could match direct and indirect object. Several possible cases may exist:
   - Empty strings (“”) are matched to $nothing (#-1)
   - Ojbect strings (e.g. #127) are matched to objects (if it exists of course)
   - “me” or “here” object “descriptions” are matched to the obvious (i.e. the player object or the room object).
   - Otherwise, the server will try to match object “descriptions” to any object name that is either on the player or in the room.
5. Finally, the server then tries to find the verb on each of the following objects:
   (a) the player who typed to command
   (b) the room the player is in
   (c) the direct object (if it found one above)

[5] http://tecfa.unige.ch/guides/MOO/ProgMan/ProgrammersManual_8.html
[6] In the MOO programming manual those “items” are called “words”

(d) the indirect object (if it found one above)
In order to match, all the following conditions must be true:

(a) the verb "description" found above matches one of the names for the verb on the object

(b) the direct and indirect object types are allowed by the corresponding argument specifiers for the verb

(c) the preposition string is matched

7.4.2 Verb argument specification

Before you program a verb you must create it (i.e. add it to and object), specify its arguments and define its permission. Typing help @verb[7] will give you all the information that is below. In order to create a verb you nee the `@verb` command. It's general syntax is:

```
Syntax:  @verb <object>:<verb-name(s)>
         @verb <object>:<verb-name(s)> <dobj> [<prep> [<iobj>]]
```

@verb adds a new verb with the given name(s) to the named object. If there are multiple names, they should be separated by spaces and all enclosed in quotes:

```
@verb foo:"bar baz mum*ble"
```

The direct and indirect object specifiers (`<dobj>` and `<iobj>`) must be either 'none', 'this', or 'any'; their meaning is discussed in the LabmdaMOO Programmer's Manuel (sections 2.2.3[8] and 3[9]).

- 'none' means that the argument is missing
- 'this' means that the value of the argument is the same as the object on which the verb is found
- 'any' means that the value of the argument can be anything.

The preposition specifier (¡prep¿) must be either 'none', 'any', or one of the following prepositional phrases:

```
with/using
at/to
in front of
in/inside/into
on top of/on/onto/upon
out of/from inside/from
```

[7]http://tecfa.unige.ch:7777/help/@verb
[8]http://tecfa.unige.ch/guides/MOO/ProgMan/ProgrammersManual_7.html
[9]http://tecfa.unige.ch/guides/MOO/ProgMan/ProgrammersManual_8.html

```
over
through
under/underneath/beneath
behind
beside
for/about
is
as
off/off of
```

A prepositional phrase with more than one word must be enclosed in quotes ("")). All three specifiers default to 'none'.

With the @verb command, it is also possible to specify the new verb's permissions and owner as part of the same command (rather than having to issue separate @chmod/@chown commands)

```
@verb <object>:<verb-name(s)> <dobj> <prep> <iobj> <permissions>
@verb <object>:<verb-name(s)> <dobj> <prep> <iobj> <permissions> <owner>
```

`<permissions>` are as with @chmod, i.e., must be some subset of "rwxd". They default to "rd" (specifying "w" for a verb is highly inadvisable, see section 7.7 for details). The owner defaults to the player typing the command; only wizards can create verbs with owners other than themselves.

You may also use "tnt" in place of "this none this" for the dobj prep iobj arguments. "this none this" is used to indicate non-command verbs, since the parser can't possibly interpret a command as "this none this". For these verbs, the permissions default to "rxd"; the "x" bit is set so that they can be called from other programs. (If they couldn't be used as commands, and they couldn't be called from programs, they wouldn't be good for anything!)

Changing verb arguments: Sometimes you need to change an argument specification, either because you misused the @verb command or because you figure that you need it differently. Use the @args command:

```
Syntax:  @args <object>:<verb-name> <dobj>
         @args <object>:<verb-name> <dobj> <prep>
         @args <object>:<verb-name> <dobj> <prep> <iobj>
```

Changes the direct object, preposition, and/or indirect object specifiers for the named verb on the named object. Any specifiers not provided on the command line are not changed. The direct and indirect object specifiers (`<dobj>` and `<iobj>`) must be either 'none', 'this', or 'any' (see above). The preposition specifier (¡prep¿) must be either 'none', 'any', or one of the prepositional phrases listed in 'help prepositions'.

Here is a transcript that shows an example on how you look at a verbs arguments and how you change that.

```
>@d me:grab
  6 #1324:grab                  MooBoy (#1324)       r d    any at/to any
-------------------------------- finished ---------------------------------
>@args me:grab any none none
Verb arguments changed.
>@d me:grab
  6 #1324:grab                  MooBoy (#1324)       r d    any none none
-------------------------------- finished ---------------------------------
```

7.5 Variables and Values

7.5.1 Values (by Thwarted Efforts)

Acknowledgement: This text is a formatted version of a message posted to *Teaching on World-MOO by Thwarted Efforts.

There are 5 types of MOO values. They are:

```
Type         Example         What typeof() returns
---------------------------------------------------
Number       3, 56, 1245     NUM
Objects      #0, #-1, #200   OBJ
Strings      "x", "foobar"   STR
Errors       E_PERM, E_NACC  ERR
Lists        {1, "bone", {}} LIST
```

Numbers Numbers are represented just as they are anywhere else, literally. MOO numbers do not have delimiters in them, such as commas (,). An optional minus sign (-) can preceed the number, making it negitive. MOO stores numbers as 32 bit signed integers, meaning it can understand any number from -2147483648 to 2147483647, big enough (small enough) for most purposes. These two values are stored in $minint and $maxint for programing convience. Note that the properties of signed integers include the fact that $minint - 1 = $maxint, meaning it wrapps around... $maxint + 1 will wrap around to the negative numbers.

Objects Objects in MOO are represented by a pound sign/hash mark (#) followed by a literal number. Object numbers behave the same way numbers do, there is a max object number that can be represented, and a minimum. (This is just to say that there are limits). Object numbers represent a named object in the MOO database. Not all object numbers are valid tho, a valid object being one that 'exists' (see help create()) in the database. Object numbers can't have math operations performed on them. Object numbers are not numbers, they are a constistant way to refer to certain sections of the database.

Strings Strings are MOO values that can contain any printable character. Strings are represented by their delimiting characters, the double quote ("). Strings can be of any length. Certain

operations can be performed on strings; concatenation can be performed with the addition operator (+). There are a number of builtin functions that handle strings. To include a double quote in a string, put a backslash (
) before it, like
", so, "
"" is a string that contains a single double quote, and """ is a syntax error. This is called "backslash escaping" a character. Any character can be "escaped" but it only makes sense to escape the double quote and the backslash.

Errors Errors are a MOO type that are used when an error occurs. They are just like any other type and you can use them yourself in a program to represent errors and to pass between verbs (or whatever you want). Errors have both a short name, which is the actual error value, and a string value, which is a more descriptive form of the error. The value and string are not interchangable (E_PERM != tostr(E_PERM)) A string containing the error description can be found by using the tostr() builtin. Builtin functions return errors (how those errors behave is dependant on the d bit of the verb... verbs are to come later). All errors have E_ before their name. For a list of errors, see help errors While E_NONE means no error, it is still an error type.

Lists Lists are like arrays in other languages. A list is a set MOO types inclosed in curly braces (). Any MOO type can be in a list, even other lists. Lists have a length, which can be found by using the builtin function length(). A list can have no elements in it, which is , and is called the empty list (sometimes called epsilon). length() == 0. You can find out what a certain element in a list is by using a number in square brackets ([]) after the list. 1,2,3 is list of length three, all three elements are numbers. 4, #4, E_PERM, is list of four elements, the fourth being an empty list.

7.5.2 MOO variables

Acknowledgement: This text is a formatted version of a message posted to *Teaching on World-MOO by Thwarted Efforts.

Variables are named slots that can be used to store a value. The names of variables always begin with a letter or an underscore (_), and can have any number of of letters, numbers, or underscores after that. Variable names are case insenitive, meaning the variable named FOO is the same as the variable named foo. We can use eval to test this:

```
;;foo = "bone";  return FOO
=> "bone"
```

Variables can not be named after any of the moo keywords, namely, while, for, return, in, endwhile, endfor, if, endif. Variables also can't be named the same as the Error values.

Variables are expressions that return the value they represent. They can take on any value at any time. Once a variable is created, it exists for the entire verb execution, but once the verb ends, the variable is destroyed for that execution cycle. Variables do not keep their values or even existance across verb calls (exceptions exist for the builtin variables) or across verb invocations. Variables can be operated on just like all other MOO values, but only using the rules of the same type with which that variable holds.

```
x = #4;
return x + 3;
=> E_TYPE  (you can't add an object and a number)

x = 4;
return x + 3;
=> 7 (you can add numbers)
```

Variables are often used to store the result of some expression. If an expression's result wasn't assigned to a variable, then the expression's result would be tossed on the floor and forgotten about.

Variables that are defined return E_VARNF (Variable not found). This will cause a traceback if the verb is +d, or the variables value will be E_VARNF if the verb is !d.

```
foo = bar;
```

This assigns the value of the variable bar to the variable foo.

```
foo = foo + bar;
```

this sets foo equal to the result of foo plus bar (what exactly foo + bar equals is dependant on the types)

foo + bar is not the same as foobar. foobar is an entirely differnt variable name.

It's good programming practice to use descriptive variable names. It's much easier to understand your code 5 months after you wrote it and it's getting an error if you can tell what each variable if for by looking at it's name.

```
for x in (players())
```

If the body of the for is big, you will find yourself wondering what x exactly holds. Its much clearer to write:

```
for dude in (players())
       or
for loser in (players())
```

it makes for better readability

The MOO sets aside some variable names for you, that contain certain things about the run time enviroment of your verb. The player variable holds the object number of the current player. args is the arguments that were passed to your verb. dobj and iobj hold the direct and indirect object specifiers on your verb, and dobjstr and iobjstr hold the strings that the player entered for the direct and indirect objects. prep holds the entered preposition. 5 variables also exist to make the results of the typeof() functions easier to read, they are STR, NUM, LIST, OBJ, and ERR. typeof() returns a number, and these variables can be used to compare, allowing greater readability than comparing to numbers.

```
if (typeof(1) == NUM)
```

is more readable than

```
if (typeof(1) == 0)
```

(note ;NUM => 0, ;STR => 2, in case you want to find out the values of these variables.)

Two other variables are this and verb, which contain the object number that the currently executing verb is on, and the name of the currently executing verb. Any builtin variable that is empty will contain "" if it's type is a string, and $nothing (#-1) if it's type if susposed to be an object.

7.6 Expressions, statements, commands, oh my! (by Thwarted Efforts)

Acknowledgement: This text is a formatted version of a message posted to *Teaching on World-MOO by Thwarted Efforts.

MOO programs are made up of statements. All statements end in a semicolon. Statements also can include other statements, in the body of a MOO command.

```
return;
```

that's one statement

```
for x in [1..10]
  y = y + x;
endfor
```

This is a statement within a statement. The for loop is a statement, and the 'body' of the for loop contains a statement.

```
fork (0)
  foo:bar();
endfork
```

This is also a nested statement, one statement in a the body of the fork, and the fork counts as a statement.

Statements are made up of commands or expressions. Commands tell the moo to do something. Commands do not return values. Commands can't be nested (but statements can). Commands can take expressions as arguments, but commands can't be used as arguments. Expressions return a value. Expressions can take arguements, and expressions can be nested (expressions within expressions)

The moo commands include: while, for, if, return.

An expressions is anything that returns a value: all the builtin functions, any MOO value, any operations on those values.

Examples of statements are:

```
0;
```

this evaluates the expression 0, and then throws the result away.

```
1+2;
```

this evaluates 1+2, and throws the result away.

```
foo = 1+2;
```

This is made up of two expressions, 1+2 and a variable assignment. foo is a variable name. 1+2 is evaluated, and the result is assigned to the variable foo. foo is evalualated the result is thrown away.

```
foo = bar = baz = 1;
```

This is a nested expression, 1 is evaluated, and assigned to baz. baz is evaluated and the result is assigned to bar. bar is evaluated and the result is assigned to foo. foo is evaluated and the result is thrown away.

```
;
```

This is the null statement. It returns 0.

```
while (1)
   foo:bar();
endwhile
```

this is a statement made up of a command. the while command takes 1 argument, an expression that returns a number. In this case, the expression is 1, which returns 1. While does not return anything. It's a syntax error to assign the result of any command to anything. x = while (1) is invalid.

7.7 Permissions (by Defender)

Acknowledgement: This text is a formatted version of a message posted to *programmers on (now defunct) Worldmoo on Sat Mar 4 07:50:23 1995 PST by Defender (#160)

1. Every object, verb, and property has an owner.

2. Verbs are executed with a 'set of permissions', or a flag indicating which player owns the task. Any verb you write will normally run with your permissions, which means it can affect things you own. Wizards can use a built-in function called set_task_perms(), documented, to change the permissions of a particular verb's execution. This doesn't change the verb ownership, nor does it change permissions for other verbs. Normally you don't need to worry about that.

3. A verb with the permissions of the owner of something, or a wizard's permissions, can affect that something (read its value or write to it). For instance, you can create verbs and properties on objects you own, because the creative verbs run with your permissions.

4. Everything (objects, verbs, and properties) have a set of possible 'permission bits', which control access to them. Two of these potential bits are always R and W (for Readable and Writable). For verbs whose permissions aren't the owner and aren't wizardly, these flags are important.

5. If the R bit for something is set, any set of permissions can read it. For example, if a verb you write has its R bit set, anybody can read the verb code. If it is not set, the object/verb/property is said to be "E_PERM'ed", referring to the error code E_PERM (Permission Denied), which will be returned upon attempts to read the value. Unless you are deliberately needing to hide some information, it is good practice to make things readable by setting the R bit.

6. If the W bit for something is set, any set of permissions can write to it. For example, if you have a property whose W bit is set, anybody can set the property value to anything they want, or remove it altogether. The W bit is almost never set, and should NOT be set for verbs in particular.

7. What reading and writing mean depends on what you are dealing with.

```
            WHAT READABILITY MEANS      WHAT WRITABILITY MEANS
Objects:    You can use verbs() and     You can define your own verbs
            properties() directly to    and properties on the object,
            read the list of verbs and  which will be owned by you.
            properties directly.
Verbs:      You can use verb_info(),    You can change the arguments,
            verb_args(), and            names, and program code for the
            verb_code() on the verb,    verb in question, using @edit
            and you can @list the code. for example.
Properties: You can see the permissions You can change the permissions
            and value for the property. and value for the property.
```

7.7.1 More on Permissions

Brief review of permission bits:

Objects - rwf

r anyone can get a list of verbs and props on your obj. This doesn't mean they can actually read those verbs or props. (That is handled by the permissions of those verbs and props).

f fertile

w if the object is writable, this means that anyone could add or remove verbs or props from your object, even if your object is not readable.

Verbs - rdxw

r the verb is readable, anyone can @list your verb

d debug bit. This means if verb hits an error, you get a traceback.

x can the verb be called from other verbs (executable)

w the verb is world writable. This means that anyone can change your verb code, but it still runs under the original author's permissions. This is almost always a BAD THING (tm).

Properties - rcw

r the property is world readable.

w this means that anyone can write to your property (ie. change the value of your prop). Generally, w permission bits, whether on objs, verbs or props, is not a great idea, though of the 3, a +w prop is probably the least problematic, though I suppose it depends on how much of a problem you think someone changing the value of your prop might present. ;-)

c this is perhaps the trickiest, (initially) of the different permission bits. If a prop is +c, anyone who creates a @kid of your object, will own the property. This means that they can directly change it, but also means that your verbs will no longer have permission to change it. So, the owner of a kid could change the prop with eval, or change the prop using @set. If you want YOUR verbs to be able to change the property, then you want so set the prop !c, so that you will be the owner of the prop for any kids. If anyone needs or wants any clarification on any of this.

Chapter 8

Issues in MOO Programming

8.1 Features

Stolen from MOOtiny (thanx Jeni), but don't believe it works at **TECFAMOO** , yes this section is also under construction ....

```
Features are objects that provide you with commands not covered by the
ordinary player objects.  The advantage of using features is that you
can mix and match the things you like; whereas if you like a command
that's defined on a player class, you have to also get all the commands
it defines, and all the commands its ancestors define.

Commands:
@features          -- lists your features
@add-feature       -- adds a feature to your list
@remove-feature    -- removes a feature from your list
@prefer            -- allows you to use one verb in preference to others


Features are often the first thing a programmer programs, because they're
fun and easy to do.  However, they are not seen as valuable assets
to the moo community for the most part.  The top-level generic feature
is $feature.

Verbs on features cannot adjust properties on the player, but they are
useful for scanning, surveying, and producing tons of spam.  They have
to be +x (use @chmod to do this), because they are called by other
verbs.

Note to programmers: In order to be available for general use as a
feature, an object must have a verb or property named "feature_ok"
```

```
which returns a true value.  When 'help <fo>' is used, the .help_msg
and the verb documentation of all verbs in the .feature_verbs property
on the fo are given.

When a feature is added to a player's features list, feature:feature_add
is called, if it exists, with the player in question as its argument.
Likewise, when a feature is removed, feature:feature_remove is called.

Commands (don't work at \tecfamoo I think):
@feature_ok        -- sets the .feature_ok property on the fo to true.
@set_feature_verbs -- sets up a list of feature verbs.
```

8.1.1 Help on Feature Object Verbs

To add help on your FO verbs: Add a line or two of comment to the top of each verb that users would call directly. Recompile the verb with the comment. You then need to set the feature verbs of your feature by using an eval:

```
;#<your-fo>:set_feature_verbs({"verb1", "verb2"})
```

Note if you have a verb with more than one name, only add one of the names in when you set the feature verbs).

8.2 Security

8.2.1 Writing Secure Verbs (by EricM)

Acknowledgement: This text is a formatted version of a message posted to ??? on BioMOO by:

```
Copyright 1995 Eric Mercer
last updated 7/11/95
EricM @ BioMOO, Diversity University, others....
mercer@caltech.edu
```

What verbs need permissions checking? Any verb that either

1. alters the state of the object in a way that only a limited number of people should be able to do, or

2. provides information that only a limited number of people should have access to.

What are the main considerations in choosing a format for your security? The main considerations are 1) is the verb +x, and 2) can the verb be called from the command line (ie. does it have args other than this none this). Note that these are not linked and you can have +x command line verbs, for instance.

What is the general structure of a permissions test? Verbs should start with a few lines of comments (describing what they do, etc.) followed by:

```
if (perms_test_fails)
   return failure_report;
endif
....->rest of verb
```

where "failure_report" should be 'E_PERM' for +x this none this verbs, and 'player:tell("Sorry, you don't have permission to do that.")' for -x command line verb). For +x command line verbs, add the player:tell line before returning E_PERM.

Replace "perms_test_fails" with one of the constructs given later in this note.

What are valid calls to +x, this none this verbs? I think the following list is complete. I've also given an example of a typical "perms_test_fails" line for each, but note that you'll often want to use some combination or hybrid.

```
1. Calls from other verbs on the object itself.
     type:object-based security
     example: if (caller!=this)
2. Wiz-permed calls
     type: permissions-based security
     example: if (!$perm_utils:controls(caller_perms(), this))  <-preferred
              if (!caller_perms().wizard)  <- less flexible
3. Calls from verbs owned by the same programmer
     type: perms-based security
     example: if (caller_perms() != $code_utils:verb_perms())
     note: Gives a lot of power to the programmer, but sometimes useful.
           Note that we DON'T use the object number of the programmer,
           because this will be different if someone ports the object.
4. Calls from verbs owned by the object owner
     type:perms-based security
     example: if (caller_perms() != this.owner)
     note: rarely useful, and not at all useful on generic objects
5. Calls from objects owned by the programmer
     type: object-based secutiry
     example: if (caller.owner != $code_utils:verb_perms())
     note: only useful in some special cases
6. Calls from "permitted" objects (generally stored as a list on a property)
     type: object-based security
     example: if (!(caller in this.permitted_callers))
     note: useful only in some specialized circumstances
```

What is the most useful test for -x command line verbs?

```
if (!$perm_utils:controls(player,this))
```

This allows the object owner, additional owners, or wizards access to the verb.

Why is it unacceptable to test "player" for security on +x verbs? I'll give an example. Loro the lazy wizard writes a +x verb that can recycle any object and tests permissions with "if (!$perm_utils:controls(player,this))" at the verb's beginning. Semli the sneaky programmer builds an object and adds a "tell" verb to it (ie. a verb that gets called any time someone in the same room speaks). The "tell" verb calls Loro's +x verb and tells it to recycle all of Loro's objects. Semli puts the object in Loro's room...and Loro gets a nasty surprise after connecting. Neato eh! Note that "player" will be the person speaking (Loro in this case), because "player" is set to whoever initiates the action, and can only be changed by wiz-permed verbs. Generally, it stays the same from the task's start to it's finish. Now, if Loro had tested caller_perms(), then Semli's call would have been caught as one that did not have permission to be recycling objects. Got it?

What's the most secure test for +x non-command line verbs?

```
if (caller != this)
```

When in doubt, use this one. It's the least flexible but the most secure.

How do I do permissions-based security for a +x command line verb? The problem here is that you can't test caller_perms() on a command line verb, since the perms will be #-1. Note that this isn't a problem for object-based security, since "caller" for a command line verb will be the same as "player." To test security on a +x command line verb, replace "player" in a construct like:

```
if (!$perm_utils:controls(player,this))
```

with an expression that will handle both command line and verb calls:

```
if (!$perm_utils:controls( (valid(caller_perms())?caller_perms()|player) ,this))
```

Note that if caller_perms is #-1, then "player" is used, otherwise caller_perms is used.

Why are +x verbs called using pass() a special case? One of the great things about MOO code is that the object-oriented nature lets you "cover" verbs by adding verbs of the same name on child objects. These can handle special cases, but otherwise simply pass(@args) down to the verb on the parent object instead. The problem is that if you use only perms-based security, this call (a legitimate one) will fail. Let's say you are testing with

```
if (!$perm_utils:controls(caller_perms(),this))
```

which is generally very reasonable. If someone makes a new generic as a child of your verb, then caller_perms() will be that person, who is unlikely to "control" the object. The solution is to use a combination object-based and perms-based test such as:

```
if ( (caller==this) || !$perm_utils:controls(caller_perms(),this) )
```

What sort of combination security tests are reasonable? Here's an example for us to dissect:

```
if ( (caller!=this) && !$perm_utils:controls(cp=caller_perms(),this) &&        (cp!=$code_ut
```

What's going on here? First, there's a test to see if caller!=this. That gives access to calls from other verbs on the same object and to calls via pass(). Then we check with $perm_utils:controls, which gives access to the object's owner, any additional owners, and to wizards. Finally, we test if the caller_perms are the same as the perms of the verb running (ie. the calling verb was written by us). This allows us to design objects that interact with each other. Note that this is a special circumstance, but one that's not terribly uncommon (eg. a class of objects and a feature object that interacts with them).

8.3 Error handling

(to be done !!) The 1.8 MOO Server has an improved error handling mechanism, no documentation here (yet), in the meantime see the MOO programmer's manual: http://tecfa.unige.ch/guides/MOO/ProgMan/ProgrammersManual_35.html[1]

[1] http://tecfa.unige.ch/guides/MOO/ProgMan/ProgrammersManual_35.html

Chapter 9

[TECFAMOO] Programming, What's different at TECFA ?

(under construction, really !! June 1 96)

9.1 [TECFAMOO] Changes due to server upgrades

Those should be standard on all well maintained MOOs (i.e. check if your server is running under server version 1.8x or later) by typing '`@version`'.

9.1.1 Numeric Verb Indexes

(From: 18 on *General

One big change is the use of numeric index rather than string index. Before you could verb_info(obj, "index") where index was a number. Now you can directly pass the number rather than a string representing the number.

BUT, numeric index starts a 1 (and goes up to length(verbs(obj))) rather than starting at "0" like string index did. I strongly encourage you to switch to numeric index and to change your code to support numeric index rather than string index.

9.2 [TECFAMOO] Local Core

Wizards in any MOO can make global reference to an object by corifing it; example `$room == #3`, you probably never do `$object_utils:isa(foo, $room)` rather than `$object_utils:isa(foo, #3)`, so that if `$room is #34372537` your code will still work. Unfortunately only wizards can define those core reference (that's how the `$`... reference are called).

The compiler parse `$foo` exactly like `#0.foo`, so adding a reference is just adding a property to `#0`.

At **TECFAMOO** , all programmers have access to the LOCAL CORE, this local core is different of the "real" core in the way that any programmer can add his/her references and that it uses a different prefix.

The prefix for local core references is `&`:

```
&foo == $local.foo
```

Any programmer can add a local core reference through the use of the `@local-corify` command.

```
Usage:  @local-corify <object> as <propname>

Adds <object> to the local core, as &<propname>
```

So, instead of telling people to add feature `#345235`, `@local-corify #345235` as `&super_cool_feature` and tell people to add the `&super_cool_feature` feature instead You will see, it's much easier to remember.

9.3 [TECFAMOO] Blocked Verbs

Janus added support for blocked verbs (aka non-overridable verbs). A blocked verb is said to be '+b', a non-blocked (the default) is said to be '-b'.

You can change the b bit of a verb by using `@chmod` or `set_verb_info()`. You can check the b bit of a verb through `verb_info()`.

Basically when a verb is made +b, the following happens:

- it prevents to add (on the object or its descendants) any verbs that have in their names one of the names of the verb you made +b.
- it prevents to add (to verb on the object or its descendants) aliases that are one of the names of the verb you made +b.
- it prevents objects to be chparented in the following situation. Any object (or a descendant of the object):
  - has a verb, of which one of its names is one of the names of the verb you made +b (on the potential new parent)
  -

Technical note: The 'b' flag for verbs behave a bit like the 'c' for properties.

Also note:

- When you make a verb +b on an object, if that object (or its descendants) has a verb have which one of the names is one of the names of the verb you make +b, there will be no problem and no warning. It will NOT make any other verb (than the one one you make +b) +b. If you want to be sure no descendants has a verb which names ... you have to use

```
@check-b*locked-verbs <object>:<verb_I_want_to_make_+b>.
```

That means you can also make +b verbs on ancestors of objects that have those verbs +b.

- The support for the 'b' flag requires extensive checking when adding a verb, renaming a verb or chparenting an object. This overhead can cause lags in some cases.
- The support for blocked verbs is done COMPLETELY in-db without any server hack.

`E_NACC` (Move refused by destination) is raised by `chparent()`, `add_verb()` and `set_verb_info()` if chparenting an object, adding a verb or renaming a verb would cause to override some blocked verbs defined on the object or its ancestors.

9.4 [TECFAMOO] Multiownership and controls

I would like to ask every programmers to change their code so their verbs DO NOT rely on the existence of the .owner property. Programmers should use the .owners property (yes just one little more s but that make all the difference); the .owners property is a list of objects.

Better than using the .owners property, use the standard security checks:

```
<foo>:controls(<bar>) => true if <foo> controls <bar>
<foo>:controls_property(<bar>, <prop>) => <foo> controls <bar>.<prop>
<foo>:controls_verb(<bar>, <verb>) => <foo> controls <bar>:<verb>
<foo>:share_owners_with(<bar> => true if one or more of <foo>'s
                                 owner also controls <bar>.

[e.g. if (foo:share_owners_with(bar)) replaces if (foo.owner == bar.owner)]
```

If you need to print a nice list of all the owners of `<foo>`, just use:

```
$string_utils:nn(<foo>.owners)
```

If you want to get a list of all the verbs you need to change, type:

```
@grep .owner in me.owned_objects
```

Please if you have questions, suggestions, comments, need help to fix your code, don't hesitate to mail me (Janus).

P.S. for those who thinks it's rude: do you prefer your verbs to traceback and immediately know where the bug is, or do you prefer to have your verbs behave weirdly and give bad/wrong results? :-)

Chapter 10

[TECFAMOO] The World-Wide Web Interface

The WWW Interface at TECFA is based upon the implementation at MOOtiny which in turn is an expansion of Sense Media's Chiba.

The principal author of the WOO interface is Jenifer Tennison[1] from the Artificial Intelligence Research Group[2] (Department of Psychology, Nottingham University), (email: (jft@psychology.nottingham.ac.uk[3]). MOOtiny's WOO core has been ported to TECFA by K. Block (thanx Irradiate!) and some differences do exist.

Acknowledgement: Jeni Tennison was very patient in helping me to write this chapter and by sending me detailed explanations. However, mistakes are all mine! Note that this chapter is REALLY under construction, trust me (Daniel Schneider, sept 95).

Also, do consult the on-line help at MOOtiny. You can read it via the the webbed on-line help system at MOOtiny[4].

10.1 The Low Level Implementation Layer

Currently, the core WWW interface can be considered as two layers:

(1) It uses the ordinary telnet port of the MOO server and calls first the `#0:do_login_command`, i.e. the usual verb executed before connection.

(2) if a "GET" request is encountered the request is passed to the Web Core (see subsection 10.1.1 below).

Details are given in the next subsections. Note that WOO applications programmers don't really need to read this. [more to insert here!!!]

[1] http://www.psyc.nott.ac.uk/aigr/people/jft.html
[2] http://www.psyc.nott.ac.uk/aigr/home-page.html
[3] mailto:jft@psychology.nottingham.ac.uk
[4] http://spsyc.nott.ac.uk:8888/help/web

10.1.1 What happens first to a "GET" request from a www client?

The `#0:do_login_command` is called whatever commands are used before connection as a character (or guest). So if you telnet to the moo, there are certain commands (like help, welcome, @who, connect), which you can use and which are all defined on the $login object as "any none any" verbs. As an example, the `#0:do_login_command` calls the $login:connect verb after a user entered 'connect xxx yyy'. Before authentication the user is assigned a negative number and after connection his client connection will be "attached" to his own character and further things not discussed here will happen.

So each "GET" request to the moo (i.e. a www client requesting an URL) also goes first through the `#0:do_login_command`. Command parsing is all done by $login(#10) verbs as you can see here:

```
#0:"do_login_command"   this none this
......
19: args = $login:parse_command(@args);
20: return $login:(args[1])(@listdelete(args, 1));
```

In principle, other http requests could also be parsed and answered (such as post etc) by putting verbs of that name on $login. (Post is being developed experimentally at the moment at MOOtiny by Jeny 25/9/95).

In the case of a "GET" http request the $login:get verb will be executed. This verb rebuilds the URL in some ways (more later ...) and passes the contents of 'GET ....' command to the www layer. It calls $web:hdisplay like this:

```
keep = $web:hdisplay($web:get_htext(req, rc, player));

  where:
  req is the URL, e.g.
      /key=628528948/1835

  rc is the headers received (client/referer information), e.g.
      {"Referer: http://tecfamoo.unige.ch:7777/key=628528948/1834",
       "User-Agent: Mozilla/1.1N (X11; I; SunOS 5.4 sun4m)",
       "Accept: */*", "Accept: image/gif", "Accept: image/x-xbitmap",
       "Accept: image/jpeg", ""}

  player is player no, e.g. #-451
```

A note about client headers: As well as the request for a certain page, the web client also sends headers such as identifying what the client is, sometimes asking the server to only send the page if it's been modified since the last time it got it, and so on. See the documentation on the offical http protocol[5] page for more details (e.g. study the HTTP/1.0 Internet Draft[6].

If :hdisplay doesn't do a server push (returning a positive value) the connection is closed as expected from a ordinary http server.

[5] http://www.ics.uci.edu/pub/ietf/http/

[6] http://www.ics.uci.edu/pub/ietf/http/draft-ietf-http-v10-spec-03.html

10.1.2 $web: the core Woo layer

Now let's examine $web and its key verbs (as in **TECFAMOO** of Sept 95):

```
Web (#1200) [ readable ]
  Child of Root Class (#1).
 #1200:get_mime                Irradiate (#226)     rxd    this none this
 #1200:generate_error          Irradiate (#226)     rxd    this none this
 #1200:path_to_list            Irradiate (#226)     rxd    this none this
 #1200:log                     Irradiate (#226)      xd    this none this
 #1200:do_login_command        Irradiate (#226)     rxd    this none this
 #1200:headdisplay             Irradiate (#226)     rxd    this none this
 #1200:get_htext               Irradiate (#226)     rx     this none this
 #1200:hdisplay                Irradiate (#226)     rxd    this none this
 #1200:do_url                  Irradiate (#226)     rx     this none this
 #1200:is_netscape             Irradiate (#226)     rxd    this none this
```

So the usual calling sequence is:

1. $web:hdisplay
2. $web:get_htext (can call itself with a modified URL)
3. $web:do_url

The *$web:hdisplay* verbs dumps back lines of html to the www client. If it breaks it means that its argument has been misconstructed somewhere down the line. It expects the following kind of construction:

```
 args[1][1]  htext:   a list of strings (the actual html text)
                      dumped to the www client
     [1][2]  wooer:   the calling player #
     [1][3]  headers: client headers
     [1][4]  keep:    connection open bit (0 or 1)
```

This is quite easy to understand, now let's examine the harder stuff, i.e. how the request is parsed by $web:get_htext.

10.1.3 $web:get_htext

$web:get_htext is one of the 3 key verbs in this layer. It is quite complex and a bit difficult to understand because many features have been added over time. It's task is to parse the URL and to hand a URL request in more "MOO readable form" to the `do_url` verb.

Click here[7] to see the current verb on MOOtiny. Note that you can inspect all verbs by using the object browser in the WWW interface.

$web:get_htext first does user authentication:

[7] http://spsyc.nott.ac.uk:8888/auth?object_verb_prop=%24web%3Aget_htext&options=desc&options=aliases&options=owner&optior

- If the request allready contains a key it retrieves the key from the URL and verifies if it is valid in several ways.
- If the request is the /auth page it creates an authentication key and returns an html description of the place where to connect the player that will be displayed by :hdisplay
- In the ordinary case (either a request by an authenticated player or a direct unauthenticated access to a MOO object), get_htext will synthesize the html code for display.
- It sets player to wooer. Since this is a wiz-owned verb, it means that player can be used in all verbs from then on to reference the object number of the person authed in. Without this setting of player, the variable player the has the object number of the connection that the web browser is using.

Next, the URL of the request is parsed.

Special Cases:

There are two types of special cases. In each case, $web:get_htext changes the form request for the special case into a normal url request of the type that $web:do_url can handle. Note that the syntax used in the WOO for all requests is always "GET" and not "POST", i.e. the total request is appended to the URL as in the examples below. See NCSA'a documentation[8] for further details.

The first special case is verbs that are on $web_root. These verbs appear in the menu at the top right of all pages in the WWW interface.

The second are _web verbs, which appear automatically at the bottom of most objects (lists are produced using <object>:list_web_verbs). When these are used, they produce a form. The url they are converted to consists of <object>:<verb>/arg1/arg2/ which is a standard way to get the results of a specific verb on an object.

It also substitutes those weird html character codes (i.e. "%7E" into "~" (using $www_utils:subst_hext)

So, most of the work of :get_htext is to figure out what type of object you want to access, i.e. it first hunts down "?" special requests, e.g. something like:

- /key=527433505/174?verb=who&do=Do+it%21&command=
  (the who listing)
- /key=527433505/174?verb=body&do=Do+it%21&command=
  (the body command)
- /key=527433505/1403?web_verb=help
  (a verb that sits on an object)
- /key=527433505/1659?web_verb=drop
  (same kind of verb)

If this is the case, then :get_htext calls itself with a string built of key, object:<web_verb>, arguments and next time proceeds like below.

The "normal" case:

[8] http://www.ncsa.uiuc.edu/SDG/Software/Mosaic/Docs/fill-out-forms/overview.html

If the the URL is non-"?" then $web:do_url is called. Here are 2 examples of a calling sequence with the "who" $web_root verb and the page verb that appears when you click on a player. The second time :do_url gets called as you can see:

```
 ************ $web:get_htext
{"/key=628528948/1835?verb=who&do=Do+it%21&command=",
 {"Referer: http://tecfamoo.unige.ch:7777/key=628528948/1835",
  "User-Agent: Mozilla/1.1N (X11; I; SunOS 5.4 sun4m)", "Accept: */*",
  "Accept: image/gif", "Accept: image/x-xbitmap",
  "Accept: image/jpeg", ""},
 #-452}
 ************ $web:get_htext
{"/key=628528948/who",
 {"Referer: http://tecfamoo.unige.ch:7777/key=628528948/1835",......, ""}
}

 ----

 ************ $web:get_htext
{"/key=628528948/1324?web_verb=page&args1=hello+young+man",
 {"Referer: ........"},
#-454}
 ************ $web:get_htext
{"/key=628528948/1324:page_web/hello%20young%20man",
{"Referer: ........ "}
}
```

10.1.4 $web:do_url

$web:do_url returns the actual htext that will be displayed by $web:hdisplay. It expects the following arguments:

```
 arg1 (url)      = URL, e.g. something like

      {"/1324"}     or
      {"/1324:page_web", "/hello young man"}

 arg2 (valid_key) = a flag if key is valid, e.g. 1


 arg3 (extra)     = referer/client information (see above), e.g.
      {"Referer: http://tecfamoo.unige.ch:7777.....  }

 arg4 (fetch)     = defining whether the actual htext of only the header
                    is to be retrieved.
```

Let's examine headers only vs. actual htext first: The fetch tag specifies whether the actual htext of the document needs to be retrieved, or only the headers. This is important because we use `$web:do_url` to route requests both for the server header information and for the html of the document.

See the lines in $login:get (MOOtiny's version of Sept 25 95)

```
fork (0)
  $web:headdisplay($web:get_htext(req, rc, player, 0));
endfork
suspend(3);
keep web:hdisplay($web:get_htext(req, rc, player, 1));
```

$web:headdisplay is the verb that displays the server headers wheras $web:hdisplay displays the htext to the user. We want the headers to be displayed before anything else, so we fork that off and get it calling :do_url and so on before anything else. We have a suspend(3); to make sure that the headers get there before the html, even if the html takes a very short time to be constructed. Notice that the only difference between the arguments is the 4th argument, which is the one that gets passed as 'fetch' to `$web:do_url`. In the first case, we don't want to bother with fetching the html, in the second case, we do.

`$web:do_url` first builds a html server header information that tells the client what server we are and what kind of document is retrieved, i.e. something like this:

```
{"HTTP/1.0 200 Document follows", "MIME-Version: 1.0", "Server: WOOM/1.0",
"Date: Thursday, 21-Sep-95 20:23:00 1995", "Content-type: text/html", ""}
```

One important thing is the Date: because clients use this to tell whether they can just reload something they have in their cache, or have to go and grab it all again.

Then, it generates the html (head of the page, htext ....).

:do_url can parse three types of url.

1. It looks to see if there isn't a url at all, in which case it returns the home page.
2. It looks to see if the first argument in the URL is specifying a property (it probably has a "." in it, and the thing before the "." is specifying an object somehow).
3. If these aren't the cases, it looks for:
   (a) if the first thing is a number which can be translated as a valid object number (e.g. http://.../123).
   (b) if it looks like a verb reference (has a : in it). Since these are common types of urls (all _web verbs and header verbs (those in the menu on the top left) use it), these are checked quite carefully to make sure none are missed.
   (c) if it is the name of a verb used in the header verbs, $web_root the $web_root:verb is used.
   (d) if the reference looks like anything else, even just the name of an object the player is holding.

Using this information, it gets headers and htext and parses them to return a list of htext, headers, keep, where keep specifies whether the connection will be kept open for the sakes of server push.

10.2 Summary of important WWW Core Objects (not done yet)

$web contains the core of the WWW interface

10.3 The Basic Applications Layer (not done yet)

[not much here yet, sorry]

Documentation: see also the webbed on-line help system at MOOtiny[9]

10.3.1 Webby verbs

See also help on web-verbs[10] on the webbed on-line help system at MOOtiny

(Copy of the moo help text @MOOtiny oct 29 /95)

```
The single easiest way to let people do something through the web with
one of your objects is to create a verb called <something>_web.  I
shall refer to these as _web verbs.

_web verbs are automatically shown at the bottom of the htext of
objects.  The verb name (without the _web) is given as a button, and
the verb documentation given beside it.

The verb documentation can be used to give people the opportunity to
enter arguments to the verb.  This is done by including ?<input type>?
in the documentation.  <input type> can be one of:
  txt<num> - produces a input text box, num long.
  chk<ref> - produces a number of checkboxes, one for each of the
             items defined in ref (see below).
  opt<ref> - produces a pull-down menu, with one option for each of
             the items defined in ref (see below).
ref is a reference to a verb or property which returns a list.  If the
list is of objects, then their names will be used, rather than their
object numbers.  Two useful verbs are provided on $www_utils to aid
this.  One is :reachable_players, and the other is :reachable_things.
See the help on these.

The values returned by the inputs included in documentation are passed
to the _web verb in args[1].  Args[1] is a list of strings, each of
which will begin with '/' (a URL list).  The values are given in this
list in the order which they are taken from the documentation.

NOTE:  when using chk as the input type, each checked input will be
```

[9] http://spsyc.nott.ac.uk:8888/help/web

[10] http://spsyc.nott.ac.uk:8888/help/web-verbs/

```
a separate string within the URL list.

_web verbs must be +x (since they are called through the web) and may
refer to player as the person who's using the web.  Remember, though,
that any messages to 'player', will actually go to the person on the
moo, and they may not necessarily be connected.  Therefore any
messages that you have should be printed in the html that is returned
by them.

_web verbs should return some html.  A good example of a _web verb is
$room:emote_web.
```

10.3.2 htext verbs

Each time you click on a object on the WWW interface (or when you navigate into a new room) the ":text" verb is called (normally it is inherited from core objects like #1, #3 ($room), #5 ($thing), #6 ($player) etc. So if you feel confident in code reading you might want to have a glance at those verbs sometimes. Type `@d <any object>:htext` to see from where the verb is inherited.

Bascially the ":htext" verbs generate the items for a full HTML page that are returned in a flat list of strings, generally the following:

1. An HTML header + the global menu on top of the page + the big title, i.e. something like:

   ```
   <html>
   <head>
   <title>island</title>
   </head>
   <body>
   <form method="GET http://tecfamoo.unige.ch:7777/"><select name="verb">
   <option>help<option>browse<option>who<option>uptime<option>location
   <option>body<option>home<option>news<option>mail<option>sendmail</select>
   <input name="do" type=submit value="Do it!">
   <input name="command" size=50></form><p align=center>
   <hr>
   <h1 align="center">island</h1>
   ```

   Note that all those elements are comma separated "STRINGS" returned in a list!

2. The description of the object

3. The contents of its .htext property

4. Object specfic items, e.g.

   - Location for Objects
   - Contents and obvious exists for locations

5. "webby verbs" attached to each object, e.g. something like:

```
<form method=GET action=http://tecfamoo.unige.ch:7777/key=209189437/1429>
<input name="web_verb" type=submit value="help">
This link will show you help and verb comments on this object
</form>
```

6. A footer, i.e. something like:

```
</BODY>
</HTML>
```

So if you decide to write your own :htext verb for an object or a generic, try to reproduce something along the same lines. Copy the generic verb that comes closest to your needs and adapt!

Also, make sure not to forget the "webby verbs" if there are some. "Webby" verbs by definition have a `_web` suffix in their name. (e.g. `help_web` is a webby verb, see section 10.3.1). The :htext verb must display those verbs. It uses the `:list_web_verbs` for that.

The text below is from MOOtiny's help system (dated oct 19 1995). Alternatively, you can directly access it at MOOtiny[11] It may not totally fit what we have here at Tecfa.

```
>help htext-verbs

To have almost complete control over what is returned when an object
is viewed over the web, you should program the :htext verb on the
object.

:htext verbs are passed two arguments:

The first (and most commonly used) is a list of strings which make up
the url, as they are on other woos.

The second is the information sent by the client to the MOO, containing
information about what version browser it is, what type of information
it will accept and so on.

:htext verbs are *not* passed any 'key' information (as they are on
most other woos), as you can use 'player' to know who is accessing the
verb.

:htext verbs should be +x, return html, and can refer to 'player' as
the object number of the person who is accessing the moo.  Remember,
though that any messages that you :tell to player will not be seen
```

[11] http://spsyc.nott.ac.uk:8888/help/web

```
by them over the web, so it's good to return them in the html as well.

Good html should be returned by the :htext verb.  As an aid to this,
there are two $www_utils verbs:  :do_html_header and :do_html_footer,
which may be useful.  To include _web verbs (see help web-verbs) in
your html, use object:list_web_verbs.  You can change which _web verbs
are shown by altering object:obvious_web_verbs.

The $www_utils are meant to aid the construction of :htext verbs.
Have a look at 'help $www_utils' for more information.

If you are interested in returning extra information for inclusion in
the headers (such as Content-type or Location), you may return two
arguments from the :htext verb rather than simply returning a list
of strings.  The first argument will contain the html as usual.  The
second argument is a list which contains header information such as
the mime type and so on.  There are utilities for creating these
headers in the $www_utils package.
```

10.3.3 The $www_utils package

Type 'help $www_utils' in the MOO

Part IV

TOOLS AND HINTS FOR DEVELOPPING CODE

Chapter 11

Development and Porting Hints

11.1 Basics

The purpose of this section is to give a few hints on how to get started using the available tools.

11.1.1 Editing

An important thing is to work with a MOO client suited for development (see section 1.4). Unfourtunately, not many such clients do exist.

Also, it is useful to have a client that can ship code for editing to a local full-screen editor, preferably one that supports MOO coding. At TECFA itself (not all of our wizards) we use the rmoo/emacs client. An alternative is to use various (not so good hacks) with tinyfugue or the new tkmoo-light client. Use an emacs client if you know emacs and work under Unix, else hunt down a decent client, it is worth the trouble! If you really can't try to learn the in-MOO editor. (See section 1.4.3 for a description of some clients and also watch out for Java applets that will allow local editing)

Important: To use a local editor, 'type @edito +l'. This will ship everything you edit to your editor (if your moo client allows you to do that). Type 'help @editoptions' for details At **TECFA**`MOO` with the emacs/rmoo setup we use the following setup:

```
>@edito
Current edit options:
-quiet_insert       Report line numbers on insert or append.
-eval_subs          Ignore .eval_subs when compiling verbs.
+local              Ship text to client for local editing.
-no_parens          include all parentheses when fetching verbs.
```

Here is an example:

```
>@edit #209:eat
```

If you use the in-line moo-client you will get something like this:

```
Now editing #209:eat (this none none).
>list
  1: choice = this.kinds[random(3)];
__2_ player.location:announce_all(player.name, " mange un ", choice, ".");
^^^^
```

Type 'look' to see the commands available (well some day we just might write a MOO editor tutorial too).

In the emacs client you get this in a SEPARATE window:

```
@program #209:eat
choice = this.kinds[random(3)];
player.location:announce_all(player.name, " mange un ", choice, ".");
.
```

To ship the code back to MOO, press ctrl-C ctrl-S, and don't change the first and the last line (e.g. the ".")!! See section 1.4.3 for more details.

11.2 Displaying and comparing code

Now there are a few tricks you should know about looking at code and porting it. Note that usually you must have a programmer bit in order to be able to look at MOO code.

If you start programming your own stuff, remember how to list the objects you own:

- To list the objects you own, type '@audit'
- At **TECFAMOO** '@sp' will give you a short listing (use this if you have many objects.

Below you will find a few commands that work in **TECFAMOO** and probably on most other MOOs too.

11.2.1 Listing code

1. *'@show'*: useful for getting an "overview"
   - '@show <obj>' shows the defined verbs and properties on an object
   - '@show <obj>:<verb>' shows some information about a verb
   - etc...
2. '@dump <obj> will dump the code for an object including its verbs. This is useful for porting, but read the Newbie MOO Wiz FAQ for details, and don't forget to port the parents too (and be prepared to adjust YOUR parent objects).
3. *@list*
   - '@list <obj>:<verb>' will list a verb for a given object. On some MOOs, the abbreviation '@l ...' will work.

- '@list <obj>:<verb> wo numbers' will list a verb for a given object without line numbers, useful for just porting a verb (Type in "wo numbers" just like that)

4. *@display*:is more fancy than @show. It allows you list various pieces of code attached to an object.

- Syntax:

```
@display <object>.[property]
                 ,[inherited_property]
                 :[verb]
                 ;[inherited_verb]
```

- E.g. '@display <obj>:' or the alias '@d <obj>:' will display the verbs of an object. In the following example we list all the verbs of object #5 (alias `$thing`):

```
>@d #5:
generic thing (#5) [ readable fertile ]
  Owned by Mother (#2).
  Child of Root Class (#1).
    #5:"g*et t*ake"            Mother (#2)       rxd    this none none
    #5:"d*rop th*row"          Mother (#2)       rxd    this none none
    #5:moveto                  Mother (#2)       rxd    this none this
    #5:"take_failed_msg take_succeeded_msg otake_failed_msg
        otake_succeeded_msg drop_failed_msg
        drop_succeeded_msg odrop_failed_msg odrop_succeeded_msg"
                               Mother (#2)       rxd    this none this
    #5:"gi*ve ha*nd"           Mother (#2)       rxd    this at/to any
    #5:examine_key             Mother (#2)       rxd    this none this
    #5:take_web                Irradiate (#226)  rxd    this none this
    #5:give_web                Irradiate (#226)  rxd    this none this
    #5:obvious_web_verbs       Irradiate (#226)  rxd    this none this
    #5:drop_web                Irradiate (#226)  rxd    this none this
-------------------------------- finished -----------------------------
```

- '`@display <$obj$>.`' or the alias '`@d <obj>.`' will display the properties of and object and so forth.
- If you are just interested in a single property or verb, type something like in the following examples. Note that the "," in ",aliases" or ",name" means that the property is inherited from above ("aliases" from #2) or a built-in property ("name").

```
>@d #5.aliases
,aliases                 Mother (#2)          r c    {"generic thing"}
----------------------------- finished ------------------------------

>@d #5.name
,name                    Built in property           "generic thing"
----------------------------- finished ------------------------------
```

5. An other way for looking at code, is of course to load a verb or a propriety into the editor (especially if you can ship it to a client), but be careful not to destry anything if you have the permission to do so !!

6. At **TECFAMOO** , you can use the following commands to ship listings into an editor window: @@dump, @@list and @@d*isplay. So instead of listing things in the buffer in which you talk you can get it in another Window. Use it, it is cool!

11.2.2 Comparing Code

A useful thing is to compare what verbs and properties are available for an object:

- e.g. '@verbs <obj>' will list the verbs an object has
- e.g. ';length(verbs(<obj>))' will list the number of verbs

  For longer code, dump it out and use a 'diff' tool.

11.3 Porting Code from another MOO

The basics for porting are explained in the Newbie MOO Wiz FAQ[1] and most information is taken from there.

Who do I ask for permission to port? There are a few formal and informal rules you should respect:

- In principle everything published or available over the Internet is copyrighted (e.g. web pages, MOO code, Java applets). If an object's description doesn't explicitly grant permission to port, ASK !
- If the code you wish to port has been programmed by a member of the MOO from which you want to port send him MOOmail. To list the owner, type for example `@d <object>`. It the owner is not a real player (e.g. hacker or core-wizard) ask a wizard for permission (type `@help wizard-list` to list wizards).
- If the code seems to have been ported over from another MOO, ask the person who ported it or look at the code to determine where it comes from or if it is ok to port.

Porting Algorithm

1. Make sure that your client doesn't wrap lines (`@wrap off`)
2. Try to figure out exactly what to port and how much work it might involve, then dump the objects you need.
   - First glance at the parents of the object you want to port: Type: `@parents <object>`. If the parents don't exist on your MOO you have to port them too. Reconsider porting or start porting the parents first.

[1] gopher://gsep.pepperdine.edu/00/gsep/technical/mud-moo/new-archwiz-faq.txt

- Check if the parents needed are the same. Often parents that have the same name are slightly different. In this case, either fix the parent (or ask a wizards help if you are not allowed to). A quick way to compare the parents is to compare the number of slots and verbs. (See section 11.2.2). Also note that some extra slots inherited from very high level objects (e.g. #1 alias `$root_class` may not be needed on your moo).
- Finally, try to find out if the object you want to port needs other objects (not parents) to work with. E.g. a "video camera" might need a "recorder" and a "tape" in order to be really useful. If you can't figure it out by reading the help or the description, dump the object as below. In any case, plan to port those objects too and even some parents of those.
- Apply the instructions below iteratively for each object you want to port. Starting porting the parents first (if needed) and then port the more important objects first.

3. Next, create an object with the right parent on your own MOO using the `@create` command. E.g. if you want to port a fancy room with parent $room (#3) type: `@create $room named ''Mike's Fancy Noisy Room''` or something like that. Write down the number, you will need it in the next step.

4. Next, dump the object on the MOO from which you port:
   - Type: `@dump #obj with create id=#new-obj`
   - `#obj` is the object you want to dump and `#new-obj` is the empty new object number you just created before.
   - Now save the code (if you can) on your local machine.

5. Edit the code (dump) you recorded (good luck with a bad MOO client!):
   - Remove the create from the first line of the dump since you allready created the object
   - Now scan the code for references to other objects (see above). Replace all the hard-coded numbers `#...` by correct ones on your MOO. The same applies for MOO using "local cores", e.g. the "`&`" objects on **TECFAMOO** .
   - Scan the dump for junk that is unwanted, e.g. inserted page or shout messages, truncated lines etc etc. and fix those.
   - Important: Fix things that won't work on your MOO. Consult your local documentation! E.g. on **TECFAMOO** we use (a non-LambdaCore compatible) multi-ownership scheme. Please have a look at chapter 9 for details on what is really different at **TECFAMOO** .
   - If you feel that most of the code is more or less acceptable go to the next step.

6. Uploading code to your MOO:
   - First of all you need to decide whether uploading the whole code or doing it piece by piece. I prefer doing it as a whole first and then fix things that didn't go well.

- Consult the documentation of your MOO client on how to upload. Uploading basically means dumping text into the moo. With the rmoo/emacs clients, you just copy the code you want to upload and then you type ctrl-c ctrl-y or under X-Windows you can also select the code and then use "Send X-Selection" from the menu. Some clients also may have an "upload file" button or command (E.g. on Tinyfugue read `/help upload`. In the worst case you may try to connect to the MOO via a simple telnet and somewhat paste your code into it.
- Now careful ! DON'T UPLOAD TWICE verb definitions, or you will have 2 definitions of the same verb! (guess that applies to most MOOs). To avoid defining verbs twice while uploading the same code several times comment out all those lines after your first uploading: e.g. change `@verb #xxxx:"a_verb'' this none this` to `"@verb #xxxx:"a_verb'' this none this`, i.e. add a " in front of the line.

A word of warning: Some objects (e.g. small social features or some kind of pets) can be ported quite easily, others do need good understanding of MOO programming. Still others need at least some wizard permission, though many objects that run under wizard permissions actually would run without if programmed in the right way.

11.4 Hunting down code and objects

11.4.1 Finding things

The @find verb @find is not a standard MOO server (or Lambda MOO db) verb, however it should be available on most MOOs. Ask around, maybe there is another verb that does the "trick".

```
'@find #<object>', '@find <player>', '@find :<verb>' '@find .<property>'
```

Attempts to locate things. Verbs and properties are found on any object in the player's vicinity, and some other places.

E.g. @find "player name" will identify a player and his location:

```
>@find Daniel
Daniel (#111) is at Daniel's Office (#139).

>@find :@find
The verb :@find is on   Kaspar(#85)--Frand's Player Class(#130)   Daniel(#111)
--Frand's Player Class(#130)

>@find :kiss
The verb :kiss is on   Not a Silly Social FO(#248)   AMCSA(#1371)
```

(... more to come, include programmer's FO that float around)

11.4.2 Verb calling sequences

(... more to come, include programmer's FO that float around)

11.5 Eval

11.5.1 Basic Eval (by Thwarted Efforts)

Acknowledgement: This text is a formatted version of a message posted to *Teaching on World-MOO by Thwarted Efforts.

Eval, a command avaiable to all programmers, is one of the most useful commands on the MOO. What it does is take it's arguments and 'evaulates' them, and returns the result. This has unending usefullness, from setting property values, to examining something in the database, to testing code fragments.

A short form of eval is the semicolon (;)

There are a few forms of the eval command, ;, ;;, and eval-d.

; will just evaluate the first expression that it gets. It essentially sticks a return command before that expression.

;; is just like ; except it takes more than one moo statement and doesn't stick a return command in it.. it's return value is dependant on what you chose to return. this is usually used to evaluate multi-line moo programs.

eval-d and it's cousin eval-d ; runs the code with the d verb bit off, so you won't get a traceback if your eval crashes.

The best way to explain this is with some examples

```
;2
=> 2
```

This just returns it's expression (the expression being 2) and is the same thing as a verb with one line of 'return 2;'

```
;2+5
=> 7
```

This is just like 'return 2+5;'

```
;;2+5
=> 0
```

This is just like the code '2+5;' No return value is generated (0 is the default return value).

```
;;return 2+5
=> 7
```

This is the code 'return 2+5;'

```
;length(1)
#-1:Input to EVAL, line 13:  Type mismatch
```

This eval is just like 'return length(1);' and since length(1) is an invalid expression, and returns E_TYPE, the eval tracebacks.

```
eval-d length(1)
=> E_TYPE (Type mismatch)
```

This is just like 'return length(1);' but run with the d bit unset, so length() doesn't cause it to traceback, it just returns the error value

```
;for x in (players()) if (x == #1353) return x; endif endfor
=> #1353 (ThwartedEfforts)
```

This is just like the verb code:

```
program ..:..
for x in (players())
  if (x == #1353)
    return x;
  endif
endfor
```

11.5.2 Eval Options (by Thwarted Efforts)

Acknowledgement: This text is a formatted version of a message posted to *Teaching on World-MOO by Thwarted Efforts.

A few options are avaiable to eval to make it easier to use and more informative. By default, you can't use the variables 'me' and 'here' in an eval. Before your eval is executed, the eval verb inserts the contents of your .eval_env prop. $prog's .eval_env is defined as "here=player.location;me=player" so all descandants of $prog can use 'me' and 'here' in their evals.

```
;me
=> #1353 (ThwartedEfforts)
```

You can add more things to your .eval_env prop, , I have 'su=$string_utils;lu=$list_utils; in my .eval_env, so I can use 'lu' and 'su' in my evals (which is much easier to type). One of the drawbacks of this is that if your eval crashes with a traceback, the line number count will be off from what you entered for your eval. (small price to pay)

You can also have it tell you about how many ticks and seconds your eval ate up, by setting your .eval_time prop to 1. (try ;me.eval_time = 1). This will not be accurate if your eval fork()'ed or suspended().

Unfortately, if you use .eval_time and .eval_env, your tick count will be effected by all the ticks that are in your .eval_env. This can be fixed by setting your .eval_ticks prop with the number of ticks your eval_env eats up. To find out how many ticks your eval_env takes, set your .eval_ticks to 0, eval_time to 1 and then put your preliminary set up code in your .eval_env. Then just do an empty eval:

```
;
=> 0
[used 23 ticks, 1 second.]
```

and it will tell you how many ticks your empty eval took. Use that value to set your .eval_ticks prop. Above, my empty eval took 23 ticks, so I set my .eval_ticks prop to 23. Then an empty eval will take 0 ticks, as it should, since there isn't any code. The usefulness of this is so that only the code that you want to evaluate will have the tick count, and your setup code will not be included.

11.6 Debugging Code

... is not easy in the MOO [more to come!!!]

The most important tricks are:

- set verbs to +d ('@chmod #xxx:verb +d'). This will produce a trace when an errour is enountered.
- Print out the arguments of verbs. Since lists or recursive lists are not printed you need a procedure for that: At **TECFAMOO** , use :report() with a single argument, .e.g. '@args'. Remember that errors (both syntax and logical) ocur because of non-anticipated or badly constructed data-structures !!
  You canOn every Lambda Core MOO you can also use this: <you>:tell($string_utils:from_value(args, 1, 10));

11.6.1 A few debugging tricks/common errors

Strange Prop Not Found Tracebacks?

OK, you get a traceback telling you line whatever, prop not found, but when you look at that line, you can't seem to find any property there. If that line has anything beginning with a $, (as in $string_utils, $rpg, $player, etc), whatever has a $ in front of it is a reference to a property on #0 (aka The System Object). When something is "@corified" on a moo, a property is added to #0, so when you see something like $string_utils, that means that you can also find:

```
@show #0.string_utils
#0.string_utils
Owner:        Wizard (#2)
Permissions:  rc
Value:        #20
```

So if you can't see an obvious property that is being referenced, see if the "prop" referred to is a corified object.

I set a prop +c that should have been !c, now what?

If the object in question doesn't have any kids, just @chmod object.prop !c

If the object has kids, there are several things that can help, but the easiest, and best way to handle it is to use a verb on $wiz_utils. Your fix would look something like the following:

```
;$wiz_utils:set_property_flags(#object, "property-name", "r")
```

Note that in the eval that you use, you DO want to have the name of the property and the permission bits you want, enclosed in double quotes as written above. For more info on this useful verb, type: 'help $wiz_utils:set_property_flags'.

(BTW, no, you don't need to be a wizard to use this!)

My FO doesn't work

First thing to check – ALL verbs on an FO need to be +x. (#1 reason why FO's don't work is the programmer forgot to set the verbs +x)

Bibliography

[ABELSON, 1985] ABELSON, H. SUSSMAN, G. (1985). *Structure and Interpretation of Computer Programs.* The MIT Press, Cambridge, Ma.

[Curtis, 1993] Curtis, P. (1993). LambdaMOO Programmer's Manual. Xerox Parc.