

Programmer's Guide to MOZ (Moo in OZ).

Robin Lee Powell

This manual is for MOZ (MOO in Oz) version 1.0.

Copyright © 2003 Robin Lee Powell

Permission is granted to distribute and modify as long as credit is given. See the file `license.txt` in the main MOZ distribution for full copyright information.

Table of Contents

.....	1
1 General Issues	2
1.1 Introduction	2
1.2 Localized Strings	2
1.3 Timeouts	2
2 Class Creation	3
2.1 Class Creation Introduction	3
2.2 Creating A Class File	3
2.3 Required Attributes And Features	3
2.3.1 Attributes	3
2.3.2 Features	4
2.4 Common Methods	5
3 Control Objects	6
3.1 Control Verbs	6
4 Verbs	7
4.1 Verb Methods	7
4.1.1 Verb Methods Are Always Public	7
4.1.2 Verb Method Default Arguments	7
4.1.3 Verb Method Result Records	7
4.2 Verb Record Structure	8
4.2.1 Parse Records	9
4.2.1.1 Parsing Directives	10
5 Core Classes	12
5.1 Core Methods	12
5.1.1 MozBase Methods	12
5.1.2 Storage Methods	14
5.1.3 Server Methods	16
5.1.4 Connection Methods	17
5.1.5 LanguageStrings Methods	17
5.1.6 Parser Methods	17
5.1.7 Described Methods	18
5.1.8 Located Methods	19
5.1.9 Mobile Methods	19
5.1.10 Location Methods	19
5.1.11 Container Methods	20
5.1.12 Player Methods	20

5.1.13	Exit Methods	21
5.1.14	Gate Methods	21
5.1.15	Terminus Methods	21
5.1.16	Control Methods	21
5.1.17	ClassControl Methods	22
6	Unsorted	23
7	Command Index	24
8	Method Index	25

This is the Programmer's Guide for MOZ (Moo in OZ). MOO is Mud Object Oriented. MUD is Multi-User Dungeon or Dimension. In general, a MUD is a multi-user text-based virtual environment. For information on MUDs in general, see <http://www.godlike.com/muds/> or your local search engine. For information on MOOs, see <http://www.moo.mud.org/moo-faq/>.

Oz is a multi-paradigmatic language that happens to not suck. See <http://www.mozart-oz.org/>.

1 General Issues

1.1 Introduction

UNFINISHED

1.2 Localized Strings

To deal with MOZ's requirement to be able to output to users in multiple languages, a MOZ programmer should never use Oz strings for output at all. Instead a structure called a Localized String is used. This is a record, with a feature for each language (using whatever short code is defined in the Server object, such as `en` for English). Each feature holds a string that is the string that should be output for a user that uses that language. An example:

```
string(  
  en: "A sample string.\n"  
  lb: ".i le mupli seltcidu\n"  
)
```

However, for output of strings from program code (as opposed to output of strings set by players directly, such as those stored in names and descriptions), you shouldn't enter these strings directly. Instead, you should first add the strings to the `LanguageStrings` object using the `***UNFINISHED***` command. The `tell` method on the `Player` object will treat any atom by itself as a key into the database on `LanguageStrings`, making it easy to use these strings in your code.

The reason to do this is it makes it much easier for others to translate everything in the MOZ to a new language if everything is collected on one object and easily retrievable.

1.3 Timeouts

When you call another object in Oz (or do anything else, for that matter), the program happily waits until processing is completed, and will happily wait indefinitely. This is a bit of a problem in a language where many bugs cause hangs.

While there's not much MOZ can do to stop you from causing a hang in your own code, there are timeouts on calls to other objects, so that another object being hung doesn't permanently hose your own code.

The default timeout is 500 microseconds (i.e. half a second). To alter this timeout, add the feature `activeObjectWaitTime` to the method call with the feature holding a positive integer representing the timeout you want, or `nil` to wait forever.

Note that the method making the call may also be under a timeout, so waiting 3000ms when the callee will timeout in 500ms is rather pointless.

Verb calss have the rather grotesque timeout of 6 minutes. `start` methods never timeout.

2 Class Creation

****UNFINISHED**** Here go some notes about what classes are and why you would want to create one.

2.1 Class Creation Introduction

****UNFINISHED**** Stuff about why you'd want to create classes (especially since this is the fundamental thing that makes one a MOZ programmer), stuff about class items and how to use them to control classes.

2.2 Creating A Class File

`create class named className` [Variable]

Use this command to create a class you can edit. Actually, it technically only creates an object of class `ClassControl`, which you then use to write and compile the class.

`write class className` [Variable]

Takes text until you enter "EOF" on a line by itself, and puts that text in the class file. You must have run `create class` for this to work.

`compile class className` [Variable]

Compiles a class written using `write class`. Note that if your class has certain types of bugs, this will hang. You will be able to continue your activity in the MOZ, though, and try again if you like.

2.3 Required Attributes And Features

****UNFINISHED****

Test.

2.3.1 Attributes

name A localized string containing the object's name in the MOZ.

storageRef

The `storageRef` attribute just holds an object reference record for the MOZ's central `Storage` object. `_Every_` object in the MOZ needs to talk to `Storage` at some point.

languageStringsObjectRef

Holds the global set of localized strings, which all system object should use.

hasProperName

A boolean declaring if the object's name is a proper name (like Alice or Bob) or a generic name (like couch or door or puppy).

verbs

A place to store the list of input verbs that this class recognizes. Note that there are special methods to deal with this structure; it should not be touched directly.

publicMethods

This is a list of methods we let *everybody* see. In particular, this list is used to give basic capabilities to an object's location, and vice versa.

The list is updated using *addPublicMethod*.

serverRef The *serverRef* attribute just holds an object reference record for the MOZ's central Server object.

2.3.2 Features

ozName The *ozName* feature; holds an Oz name value unique to the current object.

exports The *exports* feature holds a list of 2-tuples detailing the attributes to be handled by *toRecord* and *fromRecord*. In other words, it lists all attributes that need to be saved to disk to preserve the state of a member of this class, so it is *very* important to fill *exports* properly.

The 2-tuples are the name of the attribute and its type. Most type atoms are ignored, but some must be specially handled (i.e. object references, which must be mediated by the Storage object).

A minimal example:

```
exports: [
  storageRef#objectRef
  serverRef#objectRef
  languageStringsObjectRef#objectRef hasProperName#bool
  name#string
]
```

A more complicated example, with multiple inheritance:

```
exports: {Append
  {Record.toListInd
    {Adjoin
      {List.toRecord exports Location.exports}
      {List.toRecord exports Mobile.exports}
    }
  }
  % Our local exports
  [ language#atom outputPort#notPersistent ]
}
```

featExports

featExports is just like *exports*, except for features instead of attributes.

A minimal example:

```
featExports: [ ozName#name capabilityDict#dict ]
```

methodList

The *methodList* feature just holds a list of the names of the methods that the wrapper should make capabilities for, i.e. the externally available methods. Note that *upgrade* should not be here, because it's handled specially.

A minimal example:


```

methodList: [
  init start stop ozName className toRecord fromRecord revoke
  hasProperName getName setName deLocalize getVerbs addVerb
]

```

A more complicated example, with multiple inheritance:

```

methodList: {Append
  {Merge
    {Sort Location.methodList Value.'>'}
    {Sort Mobile.methodList Value.'>'}
    Value.'>'
  }
  % Our local methods.
  [
    tell setLanguage getLanguage sayVerb setStorage
    reloadVerb languagesVerb languageVerb helpVerb
  ]
}

```

className The class name, stored as an atom:

```
className: 'MozBase'
```

capabilityDict

The capability dictionary for this object.

wrapper The active object wrapper procedure for this object.

2.4 Common Methods

start The start method is run everytime a new object is created, which includes when it is first loaded into the MOZ or upgraded, as well as at other times in some cases.

The start method is normally either passed the boolean *isForUpgrade* feature if the start is being run during an upgrade (which can require special processing) or nothing at all.

stop The stop method works like the start method in all respects except that it is run before the object is shut down or upgraded.

3 Control Objects

Every object that is created causes the creation of a control object so that the player can do arbitrary things to the object they just got. This section describes some details about control objects that a programmer needs to know.

Control objects are of class `Control`, which is a child of class `Mobile`.

3.1 Control Verbs

Control objects don't use the normal verbs of their class, because they need to provide functionality base on the class they are controlling.

However, we don't want every verb available on the base object to be available on the control object (you can't go through a control object for an exit, for example) and more importantly, the verbs available on the control object *must* not be available on the controlled object (we don't want to let just anybody link an exit, for example).

So, here's how you extend the control functionality of control objects made for a class you've created. It's actually fairly simple.

First, create the verbs as normal, but instead of using `addVerb`, use `addControlVerb`.

Second, and this is *very* important, make sure that the verb methods for control verbs are *not* public.

Thirdly, instead of using `self` when referring to the object being controlled, use `@controlled`.

4 Verbs

Methods that do the processing for a verb call have special default arguments that are sent to them, as well as needing to return a specialized result value.

4.1 Verb Methods

4.1.1 Verb Methods Are Always Public

Except for the special case noted in See [Section 3.1 \[Control Verbs\]](#), page 6, methods used for a verb must always be publicly accessible, or they won't be usable. This just means adding a call like this to the `init` method on the class:

```
{self addPublicMethod( method: myVerb ) }
```

4.1.2 Verb Method Default Arguments

caller The objectRef for the calling object.

player The objectRef for the player object.

language The language of the verb itself, for when the player is able to run verbs in multiple languages (i.e. `help` has language `en`, `sidju` has language `lb`).

result The result of the verb, a record that indicatats success or failure of the method, among other things. It is described thoroughly in the next section. Result must be set at some point in the verb, or an error results when the verb exits. A common code fragment at the bottom of verbs to handle this issue is as follows:

```
if {IsFree Result} then
  Result = result(
    status: success
    certainty: 1.0
  )
end
```

This assumes that if the end of the verb has been reached without Result already being set, everything must have gone well.

force If *force* is set to true, no checking should be done to see if the current object is the one the verb call was intended for: it is assume that this object is, in fact, the correct choice. Such checking would be things like name and alias matched, for example.

4.1.3 Verb Method Result Records

The *result* argument gets filled with a with a record similar to the pseudo-code one below:

```
result(
  status: success|failure|other -- default success
  certainty: float from 0.0 through 1.0 -- default 1.0
  comments: <localized string> -- default nil, only relevant to failures
)
```

There are a couple of problems that this structure is intended to address.

The first is that we want the option of delivering a specialized failure message from whichever object and method knows best what the problem was if things didn't work. The difficulty there is that many failure will be from objects that really are **not** the one the verb call was intended for, so we don't want to return their errors to the user.

The second problem is that an object might honestly not be sure if a verb call, that would in fact be successful, is meant for itself. For example, if an alias is used, or the name matches but only in a case-insensitive fashion, those matches should introduce some doubt as to whether the object in question is really the one the verb call was intended for.

The solution to these is the *certainty* field. The *certainty* field is a number from zero (0) to one (1), inclusive, which indicates how certain the verb method is that it was the intended target for the original verb call. In the case of target uncertainty, the *status* should be 'other'.

The *certainty* field is ignored if the *status* is success.

If no *status* field was set to success out of a group of verb method checks, the result records are sorted by *certainty*. The highest non-zero value whose *status* is **not** failure is called again with the *force* argument set to true. This selects, out of the methods that weren't sure they was being talked to, the method that was *most* sure it was the intended target. Verb methods should skip all *certainty* checks when *force* is set to true.

If there are only failures, the failure with the highest *certainty* has its *comments* field de-localized and sent to the player to help them figure out what happened.

Note that non-verb methods often also have **result** arguments. If so, they will often not return a *certainty* feature as part of their result record, because that sort of thing is the verb method's job to determine.

4.2 Verb Record Structure

Please note that this first section is largely not something you need to use: the **addVerb** method allows you to avoid most of the technical details. The section on Parse Records, on the other hand, is quite important.

Verb records are used to associate particular types of user input with methods on objects. This means that when you type "list languages", a verb record somewhere (on your Player object, in fact) is used to compare against that to find out what method to call (in this case, the 'languagesVerb' method).

Verb records are stored in the **verbs** attribute. The **verbs** record has features for each language that the object has verbs on, like so:

```
verb: allVerbs(
  en: <verb records>
  lb: <verb records>
)
```

The label of the record, in this case **allVerbs**, is irrelevant, as are the labels of all records in this section, unless specifically mentioned otherwise.

The verb records themselves contain one feature for each verb word (that is, the first word of input) that the object wants to accept, like so:

```
en: verbs(
  help: <verb record>
```

```

    languages: <verb record>
  )

```

Each actual verb record contains the language the verb was called in¹, and the parsing structure:

```

help: help(
  language: en
  parses: [
    helpVerbParseName(
      method: helpVerb
      endOfInput: nil
    )
  ]
)

```

The `parses` feature and its list are both complicated and unusual, and are discussed in the next section.

4.2.1 Parse Records

The `parses` list is something that is very unusual for a MUD: it allows each verb to define how its arguments are parsed, and in fact requires that each verb do so.

Normally, a MUD understand some basic linguistic structures of one language, and attempts to shoe-horn whatever the player says into what it understands. For example, it might understand the English concepts of subject, preposition, and object, and will attempt to understand all input in those terms.

MOZ, on the other hand, allows each verb in each language to define how it wishes its input to be broken up. It attempts to do this in a way that requires as little programming knowledge as possible, but it's still not exactly simple.

The `parses` feature is, in fact, a list of records. This is used so that one verb word can access different methods, depending on how the rest of the line is parsed.

Important: the label of the individual parse records, such as `helpVerbParseName` above, *must* be unique within the verb in question on whatever object the parse record is being added, as `addVerb` uses that label to decide what to override when you update the verb.

The record structure for the records inside the `parses` list is as follows: the `method` feature contains the name of the method, whatever feature is left after that feature is removed (there should be only one) is used as the first parsing directive.

In this example:

```

help: verb(
  language: en
  parses: [
    playerHelpVerb(
      method: helpVerb
      endOfInput: nil
    )
  ]
)

```

¹ Yes, this is redundant, but there are parts of the internal code that only get to see the verb record, not the entire `verbs` structure

```
    ]
  )
```

the first, and only, parsing directive is `endOfInput`, which sees if the end of the user input has been reached². This means that nothing, other than whitespace, can follow the verb word "help" for this parsing structure to match.

On the other hand, we have:

```
list: verb(
  language: en
  parses: [
    playerListLanguagesVerb(
      method: languagesVerb
      matchWord: matchWord(
        word: "languages"
        rest: rest(
          endOfInput: nil
        )
      )
    )
    playerListHelpVerb(
      method: helpVerb
      matchWord: matchWord(
        word: "help"
        rest: rest(
          endOfInput: nil
        )
      )
    )
  ]
)
```

which matches either the word `languages`, then end of input, *or* the word `help` followed by end of input. In the two cases, different verbs are called.

In some cases, there will be parse records inside a parse directive; in this case, `matchWord` has a feature, `rest`, which is used to match everything after whatever word `matchWord` is being used to match. These parse sub-records work just like the general parse records described here, except they *cannot* be lists, they must be single records, and they should not include a `method` feature.

4.2.1.1 Parsing Directives

Directive Name	Arguments	Effect
<code>endOfInput</code>	<code>nil</code>	Matches end of character input.

² this is equivalent to the end of the line entered by the user at this time

string	any 1 atom	Bind the longest string (i.e. series of space-separated words) it can find to the atom passed to it, which is passed to the verb's method. A word is a list of anything that <code>Char.isGraph</code> returns true for. Because of this, <code>string</code> pretty much always matches the entire rest of the line. So, for example, "string:inputString" will pass the argument <code>inputString</code> to the verb's method containing the rest of the line.
stringUntil	<code>until string rest</code>	Fills <code>string</code> with words until it reaches a word that matches the word (or <i>list</i> of words) stored in <code>until</code> .
matchWord	<code>word rest</code>	The <code>word</code> argument should contain a string with the word that needs to be matched in the input. <code>rest</code> contains a full parse tree.
getWord	<code>word rest</code>	The <code>word</code> is filled with the next word in the input. <code>rest</code> contains a full parse tree.
plus	<code>first second</code>	This is the choice operator. Evaluates <code>first</code> as a full parse tree. If the parsing of <code>first</code> succeeds, returns that. Otherwise, tries to parse with <code>second</code> , returning that if successful. Otherwise fails.
multiMatchWord	<code>words wordFound rest</code>	Attempts to match anything from the list of strings in <code>words</code> . Whichever word is actually matched is given to the method in the atom named by <code>wordFound</code> . <code>rest</code> is the parse tree for everything after that word.
article	<code>wordFound rest</code>	Same as <code>multiMatchWord</code> with <code>words</code> set to the contents of the <code>articles</code> attribute on the Parser object.
mayHaveArticle	<code>wordFound rest</code>	Same as <code>article</code> , but accepts strings that <i>don't</i> start with an article as well.
bracket	<code>left right rest</code>	Matches anything entirely inside the brackets defined by <code>left</code> and <code>right</code> . Works if both <code>left</code> and <code>right</code> are words or if both are single characters, but not for a mix of the two. Note that it does not deal with nesting in any real way, and will only succeed if the first word or character in that part of the parse matches <code>left</code> and the last matches <code>right</code> , regardless of what's in the middle.

5 Core Classes

5.1 Core Methods

This is a list of methods on the core classes, for use in your programs. If it's not listed here, that's probably because the documentation is out of date, not for security reasons or anything; MOZ is a capability-based system, if you want to shoot yourself in the foot, that's fine. Please inform the author of all missing entries.

Note that this is a *very* brief treatment of the various methods; details should be gleaned from the source code.

Note further that verb methods are not listed here, because they cannot be called directly, and they can be deduced from the list of commands anyways.

5.1.1 MozBase Methods

init *ozName storageRef serverRef languageStringsObjectRef* [Method on MozBase]
 Initializes the attributes *storageRef*, *serverRef* and *languageStringsObjectRef* and the feature *ozName* to the values passed.

start [Method on MozBase]
 None Does nothing; here to be over-ridden in other classes.

stop [Method on MozBase]
 None Does nothing; here to be over-ridden in other classes.

ozName *ozName* [Method on MozBase]
 Returns the Oz Name associated with the current object in the passed variable. Would normally be named *getOzName*, but this method is used very frequently, and the value can't be changed so there would be no corresponding *setOzName* anyways.

className *className* [Method on MozBase]
 Returns the Oz Name associated with the current object in the passed variable. Would normally be named *getClassName*, but this method is used very frequently, and the value shouldn't be changed so there would be no corresponding *setClassName* anyways.

toRecord *record* [Method on MozBase]
toRecord is a very important method that runs through the elements of the exports feature and constructs a record using the information therein. This record can be pickled, saved to disk, and later loaded in with *fromRecord*.

fromRecord *record convert objectRef* [Method on MozBase]
fromRecord is the inverse operation to *toRecord*. It takes the output of *toRecord* and sets attributes appropriately. Note that this is a pure procedure: it is only called for its side effects. *convert* is the procedure to convert stored attributes of type *objectRef* into something useful, gotten from the *Storage* object. *objectRef* is used to return an object reference to the newly initialized object, with all capabilities.

revoke *method capability newCapability* [Method on MozBase]
 Revokes the current capability on the given method, assuming that the argument *capability* matches it. The new capability on that method is returned in *newCapability*.

hasProperName *hasProperName* [Method on MozBase]
 Returns a boolean declaring if the object's name is a proper name (like Alice or Bob) or a generic name (like couch or door or puppy).

setHasProperName *hasProperName* [Method on MozBase]
 Sets the **hasProperName** to true or false.

addVerb *language verb parse* [Method on MozBase]
 Adds a verb to the objects verbs record, dealing with things like over-writing the same verb parse, dealing with multiple parses of the same verb, and the fact that the whole verb record system is very baroque.

language The language the verb applies in.

verb The verb word itself ("help", "look", whatever).

parse The parse record to add. Note that this record will over-ride any other parse record for the same verb with the same label, so it's important that it be reasonably unique.

getName *name* [Method on MozBase]
 Standard variable get.

getArticledName *name* [Method on MozBase]
 Returns the object's name with the appropriate article in front of it.

getArticledStarterName *name* [Method on MozBase]
 Like **getArticledName**, but adjusts for the article being the first word of a sentence if a language requires that.

setName *name* [Method on MozBase]
 Standard variable set.

getVerbs *verbs* [Method on MozBase]
 Standard variable get.

deLocalize *inputString outputString language* [Method on MozBase]
 Returns a bare string from a localized string, based on a language argument.

Arguments:

inputString
 The string to be de-localized, in `string(lang: <string>)` format as usual.

outputString
 A normal Oz string.

language Optional, the language to de-localize into.

selfMatch *string certainty language* [Method on MozBase]

The object returns a certainty, as a value from 0 to 1, that it is the object being referred to by the string in question. The string should be localized.

Possible Certainty Values:

1.0 A perfect string match, including case.

0.9 Matches only after converting both strings to lower case (i.e. a caseless match).

addPublicMethod *method* [Method on MozBase]

Adds the given atom to the list of public methods for this object (i.e. methods for which capabilities are given out freely).

enhanceStorage *storageRef* [Method on MozBase]

Adds the capabilities on the given *storageRef* to the object's current capability set for the Storage object.

selfReference *selfRef* [Method on MozBase]

Returns a complete reference for the current object, including all capabilities. *Very insecure!*

publicSelfReference *selfRef* [Method on MozBase]

Returns a complete reference for the current object, with capabilities for only the methods in *publicMethods*.

printedList *stringList string* [Method on MozBase]

Takes the list of strings in *stringList* and concatenates them together as might be expected in a natural language string (i.e. in English, using commas and "and").

printedObjectList *objectList string* [Method on MozBase]

Like *printedList*, but the list is a list of object references, from which names are extracted.

5.1.2 Storage Methods

start *args modules serverObjFileNum realStart newMoz* [Method on Storage]

Extracts command line arguments from *args*, sets up links to external Oz modules, and if *serverObjFileNum* is passed, set the internal file number where the Server object is known to reside to that number. This only happens when the MOZ is being re-initialized.

newMoz is used to tell the method that this is the initialization of a completely new moz.

sync *None* [Method on Storage]

Syncs all MOZ objects to disk.

stop *None* [Method on Storage]

Saves all MOZ objects to disk.

info *None* [Method on Storage]

Outputs debugging information; currently all commented out.

- init** *ozName fileNumToOzName storageRef languageStringsObjectRef* [Method on Storage]
 Initializes a new Storage object, mostly using MozBase.init. *fileNumToOzName* is a dictionary that normally only contains a mapping from the number 1 to the new Storage object's Oz name. Initializes some other dictionaries.
- loadClasses** *None* [Method on Storage]
 Compiles and loads all the MOZ's .class files.
- loadObject** *fileNum objectRef init* [Method on Storage]
 This method loads an object from the disk by its number (using the *fileNumToOzName* dictionary). It returns the object record in *objectRef*.
- saveObject** *objectWrapper ozName* [Method on Storage]
 This method saves the object information to disk. Note that *objectWrapper* is just the Active Object wrapper, *not* the standard object reference object.
- createObject** *className objectRef ozName init* [Method on Storage]
 Creates an object, returning a standard object reference in *objectRef*.
- getClass** *className class* [Method on Storage]
 Returns the class code for the given *className*
- upgradeObject** *objectRef className newObjectRef* [Method on Storage]
 Upgrades the given object to the given class, returning *newObjectRef*. Note that this could be the same class name as before, but the class itself has been re-loaded in the mean time. In fact, that should be the most common type of upgrade.
- getObjectFileNum** *objectRef fileNum* [Method on Storage]
 Take an object reference record and returns the file number associated with that object reference. Not for general use!
- getServerObjFileNum** *serverObjFileNum* [Method on Storage]
 Returns the file number for the Server object. Not for general use!
- setServerObjFileNum** *serverObjFileNum* [Method on Storage]
 Sets the file number for the Server object. Not for general use!
- getObjectFromFileNum** *fileNum objectRef* [Method on Storage]
 Retrieves an object given the file number it is stored in. Not for general use!
- objectRefFromRecord** *convert* [Method on Storage]
 This is the procedure that fromRecord needs to instantiate attrs of type 'object'. Full details are in the source.
 This is so far from being for general use that it's not even funny.
- logLevel** *level* [Method on Storage]
 Set the current logging level to *level*. Logging levels are, in order from most to least verbose, *debug*, *info*, *warn*, *error*, and *critical*.
 The default is warn. For whatever level is selected, that level of log message and above (above meaning "less verbose" or "more severe") are printed.

- getConnectionModule** *module* [Method on Storage]
Returns a copy of the Connection module. That’s the Oz Connection module, *NOT* the MOZ Connection *class*. Used by the Gate and Terminus classes.
- getPickleResult** *url pickleResult* [Method on Storage]
Treats *url* as the URL to a file containing an Oz pickle, and returns the result of attempting to un-pickle that file. Used by the Gate class.
- writePickleToFile** *file value* [Method on Storage]
Writes the given value, as an Oz pickle, to the file given. The file is stripped of “/” and “\” characters, and placed under the “pickle” directory under the server’s root directory.
- getCapabilitiesFromOzName** *ozName capabilities* [Method on Storage]
Returns a full set of capabilities for the object associated with the *ozName* given.
- getObjectFromOzName** *ozName objectRef* [Method on Storage]
Returns an object reference, including a full set of capabilities, for the object associated with the *ozName* given.
- getObjectFromFileNum** *fileNum objectRef* [Method on Storage]
Returns an object reference, including a full set of capabilities, for the object associated with the file number given. Please don’t use this.
- upgradeObject** *objectRef className* [Method on Storage]
Forces an upgrade of the object in question.
- upgradeAll** *done* [Method on Storage]
Upgrades **all** objects in the MOZ. Well, OK, all the ones Storage knows about (which is everything but special user-created stuff, for which you’re on your own).
- createClass** *className controlRef result* [Method on Storage]
Creates a ClassControl object for the given *className*, after checking that no such class already exists, and returns a reference to the new object in *controlRef*.
- writeClassFile** *className string result* [Method on Storage]
Writes the class file associated with the given *className* using the *string* given as the *entire* text of the class file.
- loadClass** *className* [Method on Storage]
Recompiles the class named *className*. Not that the actual compilation is threaded off.

5.1.3 Server Methods

- init** *ozName storageRef languageStringsObjectRef* [Method on Server]
storageObjectRef startRoom
As per usual, except for *storageObjectRef*, which passes extra, better capabilities to the Server object, and *startRoom*, which passes an object reference to the player starting room.

- start** *args modules hold realStart* [Method on Server]
 As with `Storage`, except *hold*, which returns a variable that remains unbound until the server is shut down.
realStart is used to say that this is the real start call, rather than the normal one that happens when the object is created.
- stop** [Method on Server]
 Stops the server; also binds *hold* from the `start` method.
- handleLogin** *acceptObject playerRef outputPort acceptProc* [Method on Server]
 Deals with a user's attempt to log in, including creating new player objects if necessary. More details in the source.
- upgradeStorage** *newClass convert* [Method on Server]
 Storage calls this to get the server to upgrade it during an *upgradeAll* call. No user servicable parts inside.
- changePassword** *player oldPassword newPassword* [Method on Server]
 If the stored password for the login name *player* matches *oldPassword*, changes it to *newPassword*.

5.1.4 Connection Methods

- start** *socket storageRef modules parser outputStream* [Method on Connection]
 Handles the connection, reading from the TCP/IP port and passing to the parser, and then back.

5.1.5 LanguageStrings Methods

- init** *ozName storageRef* [Method on LanguageStrings]
 Nothing unusual here.
- getLanguageString** *key string* [Method on LanguageStrings]
 Simple dictionary lookup on the *languageStrings* dictionary. If the *key* passed as an argument does not exist, a blank, globally localized string is returned.)")"
- setLanguageString** *key string* [Method on LanguageStrings]
 Dictionary write on the *languageStrings* dictionary. Any languages not covered by the *string* argument are left as they were.
- resetLanguageStrings** [Method on LanguageStrings]
 Reloads all of the default language strings. Note that if new languages have been added, they will not be overwritten; only the languages that ship with the server by default will be, and only in the original strings; no newly added strings will be affected.

5.1.6 Parser Methods

- start** *storageRef modules outputPort serverStop player languageStringsObjectRef* [Method on Parser]
 Starts a new parser object. *outputPort* is the Socket object that is used for sending output to the player. *player* is an object reference to the player object.

parseOutVerb *string result* [Method on Parser]
 Just extracts the first word from the input string, which is then treated as the verb word.

parse *input* [Method on Parser]
 First tests to see if the first character, by itself, is a verb, using `matchVerbs`, then tries the whole first word, again using `matchVerbs`. If that fails, complains to the character.

runVerb *verb rest result* [Method on Parser]
 Attempts to match the input verb against any verb it can get its hands on, starting with the player object, then the player's contents, then the player's room, then everybody in the room.

This method does *not* implement the verb record parsing strategy; it calls `verbParseRest` for that.

verb contains the verb word, *rest* contains the rest of the input, and *matched* is set to true if a match was found.

verbParseRest *verbParse rest verbMethod language result* [Method on Parser]
 Implements parsing of verb records. Takes the parse segment of a verb record, and returns a record named after the verb word with the various `arg1:`, `arg2:` ... elements in it, filled according to the parse record.

verbParse The parse record for the verb.

language The language of the verb match we're working against.

result The results of the parse, as a record named after the verb word with features named according to the parse record.

eval *input* [Method on Parser]
 Evaluates the input as a piece of Oz code.¹

5.1.7 Described Methods

init *ozName storageRef name description* [Method on Described]
 Adds name and description attributes to the standard init.

getDescription *description* [Method on Described]
 Standard variable get.

setDescription *description* [Method on Described]
 Standard variable set.

deLocalize *inputString outputString language* [Method on Described]
 Returns a bare string from a localized string, based on a language argument, using the MozBase version of the same method but passing a value for the Server class. The Server information allows using the Server's default language value as a fallback.

¹ Currently isn't implemented as a verb; this needs to be fixed.

5.1.8 Located Methods

init *location* [Method on Located]
 Adds location to Described's list.

getLocation *location* [Method on Located]
 Standard variable get.

5.1.9 Mobile Methods

setLocation *location* [Method on Mobile]
 Standard variable set.

5.1.10 Location Methods

init *contents* [Method on Location]
 Adds contents information to Described's list.

addToContents *objectRef* [Method on Location]
 Adds the given object to the current *contents* list.

getContents *contents* [Method on Location]
 Standard variable get.

getContentsString *string* [Method on Location]
 Returns a string that contains a list of the names of all the objects in the object's *contents* list.

removeFromContents *objectRef* [Method on Location]
 Removes the given object from the current *contents* list.

wantToGet *newLocation origRecord newRecord* [Method on Location]
 This is called by other objects who desire to get one of the objects from our *contents* list. See [\[get on Location\]](#), page 20. *origRecord* is a minimal object reference to the object that *newLocation* wants.

wantToGive *oldLocation objectRef result* [Method on Location]
 This is called by other objects who desire to give us one of the objects from their *contents* list. See [\[give on Location\]](#), page 20. *result* is set to true if we accept the object, false otherwise.

searchByObjectName *name except objectRef result language* [Method on Location]
 Returns the object in our contents best matching the given name, else returns a standard result in *result*. *except* is used to exclude the calling object from the searching, as this will cause a hang.

announce *string except* [Method on Location]
 This is called by other objects who desire to have a string presented to the tell methods of all objects in this location (at least, those `_with_` tell methods).

The `except` argument takes an object reference and causes the string to *not* be presented to that object. This is very important, because if you call `announce` and the object making the `announce` call has a `tell` method, the `announce` will hand trying to re-enter that object. So, always put the object calling `announce` in the `except` argument!

get *fromLocation objectRef* [Method on Location]

Gets an object from another object, which must be a descendant of location.

First we call `getFrom` on the `from` location, then put the object that that call returns into our contents list.

Note that *objectRef* is limited reference to the object we want to *get*.

See [[wantToGet on Location](#)], page 19.

give *toLocation objectRef* [Method on Location]

Gives an object to another object, which must be a descendant of location.

First we call `wantToGive` on the `from` location, then remove the object that we give to that call from our contents list if the call returns true. See [[wantToGive on Location](#)], page 19.

5.1.11 Container Methods

init [Method on Container]

Merges the *inits* of Location and Thing; Location's *init* is run first.

5.1.12 Player Methods

init [Method on Player]

Runs the *init* methods for Location and Mobile (in that order). Sets *language* and *outputPort* to nil.

start *modules outputPort realStart* [Method on Player]

Sets the *outputPort* object variable.

realStart is used to say that this is the real start call, rather than the normal one that happens when the object is created.

stop [Method on Player]

Nothing special here, except during upgrades.

setStorage *storage* [Method on Player]

Just used to set the storage attribute with a new capability set. Used for wizardry and dewizardry and the like.

tell *string language* [Method on Player]

Sends a string to the user. The string can either be a single string or a list of strings. Any individual string can be either a localized MOZ string or a single atom. If it is an atom, that atom is looked up on the LanguageStrings object and the result is output.

setLanguage *language* [Method on Player]
Standard variable set.

getLanguage *language* [Method on Player]
Standard variable get.

setStorage *storageRef* [Method on Player]
Standard variable set, for storageRef.

grabInputUntil *untilString inputString result* [Method on Player]
Takes all input entered by the player until the player types the string passed in **untilString**. The input is returned in **inputString**. **result** is as per usual.
This method can only be called once every thirty seconds, to prevent malicious code from not letting the player interact with the rest of the MOZ. Further attempts to call this method will return a failure result instantly.

5.1.13 Exit Methods

setDestination *destination* [Method on Exit]
Sets the destination for this exit to the object given.

setName *name* [Method on Exit]
Sets the exit's name. Also creates verbs corresponding to the new name, so the player can use the exit.

5.1.14 Gate Methods

setDestination *destination* [Method on Gate]
Sets the destination for this exit to the object referenced by the pickle stored at given url.

5.1.15 Terminus Methods

getTicket *ticket* [Method on Terminus]
Uses the Connection module to return a ticket to itself.

writeTicket *file* [Method on Terminus]
Uses the **writePickleToFile** method on Storage to write a ticket for a reference to itself to the file given (which will be physically stored in the "pickle" directory under the server root directory).

5.1.16 Control Methods

getName *name* [Method on Control]
Generates a name based on the name of the underlying controlled object. For example, if the controlled object is named "Dead Parrot", this method will return "Control Rod For a Dead Parrot". Also updates the Control object's **name** attribute.

selfMatch [Method on Control]
Makes sure that its name is set properly, based on the current name of the controlled object, then runs the normal **selfMatch** method.

otherwise [Method on **Control**]

The **otherwise** method is privileged in Oz: it is called for any method call that doesn't match anything else. It is used here to handle the extensibility of **Control** verbs and methods based on the underlying controlled object's class.

getMethodList *methodList* [Method on **Control**]

Returns the combined methods of the **Control** object and the controlled object.

getVerbs *verbs* [Method on **Control**]

Returns **getControlVerbs** on the underlying object.

start [Method on **Control**]

Generates a perfect object reference on the controlled object (i.e. one that can never lose capabilities) using a special capability set from **Storage**, and sets its **name**.

publicSelfReference [Method on **Control**]

Calls **selfReference**: with **Control** objects, possession is ten tenths of the law.

5.1.17 ClassControl Methods

getName *name* [Method on **ControlClass**]

Generates a name based on the name of the underlying controlled class. Also updates the **Control** object's **name** attribute.

selfMatch [Method on **ControlClass**]

Makes sure that its name is set properly, based on the current name of the controlled object, then runs the normal **selfMatch** method.

6 Unsorted

- Provided information on how to upgrade a class of objects, including the case where there are new `init()` attributes on the new class, thus requiring an upgrade then a separate set-method call of some kind, probably to a temporary set-method.
- Add `objectName` or whatever to the verb record info.
- Example verb creation, including the method.
- Examples of 'fun' objects (meep, wind-up ducky, tame falcon).
- Some documentation on `Parsing.oz`, or a pointer to it
- Put a list of articles somewhere.
- Document the standard arguments to verb methods.
- Note somewhere that an `init` method should always be able to accept nothing but `"ozName"` and `"storageRef"`, because upgrades will pass only those (and fill them with `nil`, expecting `fromRecord` to fix them).
- Describe all the different `export` and `featExport` types (or, rather, copy this info from the design doc).

7 Command Index

C

compile class..... 3
create class..... 3

W

write class..... 3

8 Method Index

A

addPublicMethod on MozBase	14
addToContents on Location	19
addVerb on MozBase	13
announce on Location	19

C

changePassword on Server	17
className on MozBase	12
createClass on Storage	16
createObject on Storage	15

D

deLocalize on Described	18
deLocalize on MozBase	13

E

enhanceStorage on MozBase	14
eval on Parser	18

F

fromRecord on MozBase	12
-----------------------------	----

G

get on Location	20
getArticledName on MozBase	13
getArticledStarterName on MozBase	13
getCapabilitiesFromOzName on Storage	16
getClass on Storage	15
getConnectionModule on Storage	16
getContents on Location	19
getContentsString on Location	19
getDescription on Described	18
getLanguage on Player	21
getLanguageString on LanguageStrings	17
getLocation on Located	19
getMethodList on Control	22
getName on Control	21
getName on ControlClass	22
getName on MozBase	13
getObjectFileNum on Storage	15
getObjectFromFileNum on Storage	15, 16
getObjectFromOzName on Storage	16
getPickleResult on Storage	16
getServerObjFileNum on Storage	15
getTicket on Terminus	21
getVerbs on Control	22
getVerbs on MozBase	13
give on Location	20

grabInputUntil on Player	21
--------------------------------	----

H

handleLogin on Server	17
hasProperName on MozBase	13

I

info on Storage	14
init on Container	20
init on Described	18
init on LanguageStrings	17
init on Located	19
init on Location	19
init on MozBase	12
init on Player	20
init on Server	16
init on Storage	15

L

loadClass on Storage	16
loadClasses on Storage	15
loadObject on Storage	15
logLevel on Storage	15

O

objectRefFromRecord on Storage	15
otherwise on Control	22
ozName on MozBase	12

P

parse on Parser	18
parseOutVerb on Parser	18
printedList on MozBase	14
printedObjectList on MozBase	14
publicSelfReference on Control	22
publicSelfReference on MozBase	14

R

removeFromContents on Location	19
resetLanguageStrings on LanguageStrings	17
revoke on MozBase	13
runVerb on Parser	18

S

saveObject on Storage	15
searchByObjectName on Location	19
selfMatch on Control	21
selfMatch on ControlClass	22
selfMatch on MozBase	14
selfReference on MozBase	14
setDescription on Described	18
setDestination on Exit	21
setDestination on Gate	21
setHasProperName on MozBase	13
setLanguage on Player	21
setLanguageString on LanguageStrings	17
setLocation on Mobile	19
setName on Exit	21
setName on MozBase	13
setServerObjFileNum on Storage	15
setStorage on Player	20, 21
start on Connection	17
start on Control	22
start on MozBase	12
start on Parser	17
start on Player	20
start on Server	17
start on Storage	14
stop on MozBase	12
stop on Player	20

stop on Server	17
stop on Storage	14
sync on Storage	14

T

tell on Player	20
toRecord on MozBase	12

U

upgradeAll on Storage	16
upgradeObject on Storage	15, 16
upgradeStorage on Server	17

V

verbParseRest on Parser	18
-------------------------------	----

W

wantToGet on Location	19
wantToGive on Location	19
writeClassFile on Storage	16
writePickleToFile on Storage	16
writeTicket on Terminus	21