The TOM Tome

Pieter J. Schoenmakers
Programmers Without Deadlines

Eindhoven, the Netherlands

$Revision: 1.24 $
$Date: 2001/04/08 22:04:28 $

The TOM Tome
by Pieter J. Schoenmakers

Published $Date: 2001/04/08 22:04:28 $

Table of Contents

Preface **10**

I. TOM: The Language **11**

- 1. Getting started 12
  - Hello, world! 12
  - From source to running program 13
- 2. Expressions 15
  - Numeric constants 15
  - Operators 16
  - Local variables 18
  - Loops 19
  - More operators 21
  - Conditionals 22
- 3. Methods 24
  - Definitions 24
  - Tuples 25
  - Return values 28
- 4. Basic types 30
  - Numeric types 30
  - The boolean type 31
  - The pointer type 31
  - The selector type 31
  - The void type 31
  - The dynamic type 32
  - Object type 32
  - The id type 33
  - Tuple types 34
- 5. Classes 36
  - Inheritance 36
  - Object variables 37
    - Qualifiers 38
      - mutable 38
      - obsolete 38
      - private 39
      - protected 39
      - public 39
      - static 39
      - local 39
      - redeclare 39
  - Method overriding 40
  - Messaging `super` 40
  - Object allocation and initialization 41
  - Object destruction 43

Polymorphism ... 43
A class is not atomic ... 44
Multiple inheritance ... 45
Semantics ... 45
Messaging super ... 46
Special classes ... 47
Collections ... 48
6. Advanced topics ... 49
Blocks ... 49
Basics ... 49
The `eval` method ... 52
Conditions ... 53
Non-local gotos ... 53
Issuing conditions ... 54
Condition handlers ... 55
`unwind` ... 57
`signal` example ... 57
Glueing TOM and C ... 59
C functions for TOM methods ... 59
Selector names ... 60
Type names ... 61
External implementation ... 62
The hack ... 63
Interaction with the Garbage Collector ... 64
Method forwarding ... 64
Forwarding mechanism ... 64
Speed ... 66
II. TOM: The Libraries ... 68
7. The TOM Runtime Library ... 69
Program startup ... 69
C names for TOM types ... 69
Selectors ... 70
Message dispatching ... 71
Messaging from C ... 73
More types ... 74
struct name ... 74
trt_selector_args ... 74
enum trt_type_encoding ... 75
Functions ... 75
`byte_string_with_c_string` ... 75
`byte_string_with_string` ... 75
`trt_assign_local_var` ... 75
`trt_assign_object_var` ... 76
`trt_ext_address` ... 76

`trt_selector_args_match` ... 76
`trt_selector_named` ... 77
`trt_type_name` ... 77
`xmalloc` ... 77
8. Unit `tom` ... 78
File tom/All ... 78
File tom/Array ... 85
File tom/Bag ... 86
File tom/Block ... 87
File tom/BucketDictElement ... 90
File tom/BucketElement ... 92
File tom/BucketIntDictElement ... 94
File tom/BucketPDictElement ... 96
File tom/BucketSetElement ... 97
File tom/Bundle ... 99
File tom/ByteArray ... 101
File tom/ByteStream ... 103
File tom/ByteString ... 104
File tom/ByteSubstring ... 108
File tom/C ... 110
File tom/CharArray ... 112
File tom/CharEncoding ... 113
File tom/CharString ... 120
File tom/Condition ... 122
File tom/ConditionClass ... 124
File tom/Conditions ... 125
File tom/Cons ... 127
File tom/Constants ... 129
File tom/DCons ... 132
File tom/Date ... 136
File tom/Descriptor ... 141
File tom/Dictionary ... 142
File tom/DoubleArray ... 145
File tom/EqDictionary ... 147
File tom/EqHashTable ... 148
File tom/EqSet ... 149
File tom/Extension ... 150
File tom/File ... 152
File tom/FloatArray ... 155
File tom/HashTable ... 157
File tom/Heap ... 161
File tom/HeapElement ... 163
File tom/IntArray ... 164
File tom/IntDictionary ... 166

File tom/IntegerRangeSet ... 167
File tom/Invocation ... 173
File tom/InvocationResult ... 177
File tom/Limits ... 178
File tom/Lock ... 179
File tom/MutableArray ... 183
File tom/MutableByteArray ... 185
File tom/MutableByteString ... 188
File tom/MutableCharArray ... 189
File tom/MutableCharString ... 190
File tom/MutableDoubleArray ... 191
File tom/MutableFloatArray ... 192
File tom/MutableIntArray ... 194
File tom/MutableObjectArray ... 195
File tom/MutablePointerArray ... 197
File tom/MutableString ... 198
File tom/Number ... 198
File tom/ObjectArray ... 201
File tom/Pointer ... 203
File tom/PointerArray ... 205
File tom/PointerDictionary ... 206
File tom/Queue ... 207
File tom/Random ... 209
File tom/RandomDouble ... 211
File tom/Runtime ... 219
File tom/Selector ... 228
File tom/Set ... 229
File tom/Sorted ... 230
File tom/State ... 235
File tom/StreamBuffer ... 239
File tom/String ... 241
File tom/StringStream ... 246
File tom/Thread ... 247
File tom/Trie ... 248
File tom/TypeDescription ... 251
File tom/Unicoding ... 252
File tom/Unit ... 254
File tom/XL ... 255
File tom/archiving ... 258
File tom/behaviours ... 264
File tom/coding ... 269
File tom/collections ... 285
File tom/config ... 296
File tom/holes ... 296

File tom/numbers ... 298
File tom/streams ... 313
File tom/unique-strings ... 319
9. Unit `C` ... 322
File C/Math ... 322
File C/Std ... 325
10. Unit `too` ... 327
File too/AutoLock ... 327
File too/Connection ... 328
File too/DescriptorDelegate ... 333
File too/DescriptorSet ... 333
File too/PortCoder ... 335
File too/PortDecoder ... 336
File too/PortEncoder ... 337
File too/Proxy ... 338
File too/RunLoop ... 342
File too/Timer ... 344
File too/inet ... 346
File too/network ... 352
File too/ports ... 352
File too/Nameserver ... 353
11. Unit `_builtin_` ... 355
`Any` ... 355
`Any` ... 355
III. Reference ... 356
I. Tools man pages ... 357
tesla ... 358
tig ... 360
tug ... 361
gp ... 361
tomc ... 363
IV. Appendices ... 364
A. TOM makefiles ... 365
Basics ... 365
Important macros ... 365
Targets ... 366
More macros ... 366
Secondary GNUmakefiles ... 367
Environment Variables ... 368
B. GNU General Public License ... 370
Preamble ... 370
GNU GENERAL PUBLIC LICENSE TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION ... 370

Appendix: How to Apply These Terms to Your New Programs 374
Glossary 377

List of Tables

2-1. Numeric types 15
2-2. Important non-printable characters 16
2-3. Operators 17
2-4. Examples of >>> and -> 18
6-1. type encodings 60
6-2. C types for TOM types 62
6-3. Speed of method invocation 66
7-1. C names for TOM types 70
7-2. Selector type encodings 70
7-3. Example selector name encodings 71

Preface

This book accumulates the information that I have written about TOM in the past four years. That's exactly four years, as I happen to be writing this preface on 16 November 1999, TOM's fourth birthday.

The TOM Tome is written in DocBook. When it is finished, I expect to know a lot about DocBook. At the moment, all I know is that it appears to be a good decision.

In its current form, this book needs an editor's hand. However, my priority is to get all the information in, instead of presenting it in a pleasant way. Having all the information in one place already offers much progress over the previous situation where information was dispersed in many locations.

Eindhoven, 16 November 1999

Pieter Schoenmakers <`tiggr@gerbil.org`>

I. TOM: The Language

This first part of *the TOM Tome* introduces the TOM language and its use. We touch on the libraries, which are given a thorough treatment in the second part.

Chapter 1. Getting started

Hello, world!

This is the famous *Hello, world!* program, in TOM this time -- the line numbers are for clarity; they are not present in the actual source:

```
 1  implementation class HelloWorld
 2
 3  int
 4    main Array argv
 5  {
 6    [[[stdio out] print "Hello, world!"] nl];
 7  }
 8
 9  end;
10
11  implementation instance HelloWorld end;
```

The first line denotes the start of the implementation of a class named `HelloWorld`. The class ends with the `end;` at line 9. Between these lines a *method* is defined.

A method is a piece of code associated with an object. In this case the object is the `HelloWorld` class object. The method we define is called `main`. It returns a value of type int and accepts one argument of type Array. The name of the argument is `argv`; the code within the method can refer to the value of the actual argument through the name `argv`.

Line 6 contains three nested bracketed expressions; each expression *sends a message* to an object. The inner expression is `[stdio out]`. Here, the argumentless `out` message is sent to the `stdio` class object; this object is the *receiver* of the message. In response to this message, the corresponding method will be invoked, in this case the `out` method of the `stdio` class. Because of this correspondence, the terms *sending a message* and *invoking a method* are synonyms and they will be used as such throughout this book.

The `out` method of the `stdio` class returns an `OutputStream` object to which information can be written. This stream is usually connected to the user's terminal. Execution of the `main` method will resume when the `out` method has returned.

If we call the result of the first method invocation `x`, the second expression becomes:

```
[x print "Hello, world!"]
```

This sends to the object `x` the message `print` with a single argument, the string 'Hello, world!'. The stream object will print the string on the user's terminal and return itself. The third method invocation thus becomes:

```
[x nl]
```

In response to this, the stream object will flush any buffered output to the user's terminal and emit a new-line character. Like the `print` method, it returns itself. This value is ignored here.

In line 7, the method body ends. We did not indicate a return value, so the default value is returned: 0.

In line 11 the instances of the `HelloWorld` class are descibed. Since we have no use for them in this example, the definition can be empty.

From source to running program

The easiest way to get our program running is by creating a `GNUmakefile` with the following contents:

```
UNIT=               hello
TOM_SRC=            hello

TOM_MAKEFILES_DIR=  /usr/lib/tom/makefiles
include $(TOM_MAKEFILES_DIR)/GNUmakefile.bin
```

This `GNUmakefile` assumes that our program source is contained in the file `hello.t` and that our executable program will be called `hello`. Furthermore, it assumes that the TOM Makefiles (see Appendix A) are available in `/usr/lib/tom/makefiles`; adjust the definition of the macro `TOM_MAKEFILES_DIR` if their location on your machine is different.

If we now run `make`, output will be similar to this (what appears here has been edited to fit better):

* *[In the preceding paragraph, using <command>make</command> results in **make**, which is rather ugly.]*

```
$ make
/usr/local/lib/tom/makefiles/GNUmakefile.common:168: GNUmakefile.link: No such file or direc
building GNUmakefile.link...
if test -z "yes"; then touch GNUmakefile.link; else \
        /usr/local/lib/tom/makefiles/genlinkfile -o GNUmakefile.link \
         \
         -I /usr/local/lib/tom  -I /usr/local/lib/tom tom; \
fi
using unit tom from /usr/local/lib/tom/tom
test -f hello.u || \
        tug -u hello   -U tom hello.t
time tesla  -freadable-c :cc-pre :cc-post -1  \
        -u hello -I . -I /usr/local/include -I /usr/local/lib/tom
Number of units: 3
Loading unit tom...
Loading unit hello...
Preparing unit _builtin_...
Preparing unit tom...
Preparing unit hello...
1.94user 0.08system 0:02.82elapsed 71%CPU (0avgtext+0avgdata 0maxresident)k
```

```
0inputs+0outputs (625major+665minor)pagefaults 0swaps
touch .stamp-prepare
/usr/local/lib/tom/makefiles/libtool  --mode=compile gcc -g -O2     -I /usr/local/include -
I /usr/local/lib/tom -I . -c hello.c
rm -f .libs/hello.lo
gcc -g -O2 -I /usr/local/include -I /usr/local/lib/tom -I . -c  -fPIC -DPIC hello.c -
o .libs/hello.lo
gcc -g -O2 -I /usr/local/include -I /usr/local/lib/tom -I . -c hello.c -o hello.o >/dev/null
mv -f .libs/hello.lo hello.lo
/usr/local/lib/tom/makefiles/libtool  --mode=compile gcc -g -O2     -I /usr/local/include -
I /usr/local/lib/tom -I . -c hello-r.c
rm -f .libs/hello-r.lo
gcc -g -O2 -I /usr/local/include -I /usr/local/lib/tom -I . -c  -fPIC -DPIC hello-
r.c -o .libs/hello-r.lo
gcc -g -O2 -I /usr/local/include -I /usr/local/lib/tom -I . -c hello-r.c -o hello-
r.o >/dev/null 2>&1
mv -f .libs/hello-r.lo hello-r.lo
TOM_MAKEFILES_DIR=/usr/local/lib/tom/makefiles /usr/local/lib/tom/makefiles/libtool -
-mode=link  \
        gcc    -L/usr/local/lib hello.lo hello-r.lo   -L/usr/local/lib/tom/tom -
ltom -ltrt \
         -ldl -lpthread  -l_builtin_ -o hello
gcc -L/usr/local/lib hello.o hello-r.o -L/usr/local/lib/tom/tom -ltom -ltrt -
ldl -lpthread -l_builtin_ -o hello
```

* *[Briefly explain every command and its use.]*

Now our program has been built, we can run it:

```
$ ./hello
Hello, world!
```

Of course, since we are using make, we can change our program `hello.t` as much as we want and all we need to do to rebuild it is run `make` again.

Chapter 2. Expressions

In the previous chapter you've seen how a TOM program is built from files, which contain classes and instances, which contain methods. This chapter explains the basics of methods, what you can make them do, and how to write that down in TOM.

Everything in a method body is an expression. We'll start with the simplest kind of expression: constants.

Numeric constants

An example of a rather simple expression is `1 + 2`. When executed its value is `3`, just like you'd expect. Both the `1` and `2` themselves are expressions too. They are the simplest of expressions, and called *constants*. As with all expressions, constants have a type. The type of `1` is int.

int is one of the numeric TOM types. Table 2-1 lists all numeric types available in TOM. If you're familiar with C, you'll recognize most of them. The difference is that in TOM the precision, range and signedness of the types are fully defined: there is no difference between machines. Furthermore, the char and short C types have been replaced by byte and char respectively, for reasons which will become apparent soon.

There are several kinds of numeric constants, each with a specific type. Constants like `1`, `123456`, `0377`, and `0xff` are of type int. A leading zero, as in `0377`, denotes a number in octal notation; `0xff` with its leading `0x` denotes a number in hexadecimal notation. The case of the hexadecimal digits is ignored, thus `0Xff`, `0xFf`, `0XFF`, and `0xff` are all equal.

Table 2-1. Numeric types

type	description
byte	8 bit unsigned integer
char	16 bit unsigned integer
int	32 bit signed integer
long	64 bit signed integer
float	single precision floating point
double	double precision floating point

An integer constant suffixed with `l` or `L` is of type long. Thus `0L` is a `0` typed long. If an integer constant, which is not an explicit long, is too large to be held by an int, its type will be long. If even a long can not hold the value, an error is issued by the compiler.

A byte constant is written as a character enclosed in single quote characters. Thus, `'a'` is a byte with a value of 97; 97 being the ASCII value of the letter 'a'. The quote itself can be escaped using a '\' (backslash). Thus, `'\''` is a byte with a value of 39. To get a backslash, it too must be escaped---`'\\'`.

Not all byte values can be entered as a character constant, simply because not all ASCII values translate to printable (and typeable!) characters. Such characters can be entered by escaping their octal value. Thus, `'\041'` is the capital letter `A`.

In general, it is awkward to have to remember numeric values for often-used non-printable characters. Table 2-2 lists the shorthands of important character constants which stand for some of the unprintable ASCII values.

Table 2-2. Important non-printable characters

constant	value	name	description
`'\b'`	`0x08`	`BS`	backspace
`'\f'`	`0x0c`	`FF`	form feed
`'\n'`	`0x0a`	`NL`	new line
`'\r'`	`0x0d`	`CR`	carriage return
`'\t'`	`0x09`	`HT`	tab
`'\v'`	`0x0b`	`VT`	vertical tab

A float constant is a number which includes a decimal point, an exponent, or both. Thus `1.0`, `1e23`, and `4.2e1` are all floating point constants of type float.

Floating point constants of type double must have an exponent part and have 'd' as the exponent indicator. Thus `1d` is a floating point 1 of type double.

A floating point constant which seems to be a float, but the value of which is too large to be held by a float, is also taken to be a double. If the value of a constant is too large to even fit a double, the compiler will issue an error.

Operators

Expressions are composed of operators and operands. Operators come in three flavours: with 1, 2, or 3 operands. Each operand in turn is an expression.

Each operator has a priority. For instance, the priority of the `*` is higher than that of `+`, causing `1 + 2 * 3` to be interpreted as first multiplying `2` by `3` followed by the addition of `1`. Parentheses can be used to impose a different evaluation order: `(1 + 2) * 3` will first add `1` and `2`, multiplying the result with `3`.

You can experiment with expressions by modifying the `hello.t` program to print something different from its familiar message. Here are a few examples; remember to run `make` to have the program rebuilt after you have modified the source.

```
[[[stdio out] print 1 + 2 * 3] nl];
[[[stdio out] print (1 + 2) * 3] nl];
```

And a more daring example---notice how `print` accepts more than one thing, grouped by parentheses and separated by commas.

```
[[[stdio out] print ("0 F = ", 5.0 / 9.0 * (0.0 - 32.0), " C")] nl];
```

Table 2-3 lists all operators. Operators nearer to the top have a higher priority than those below it. In the same group, between horizontal lines, operators have the same priority.

* *[Horizontal lines have disappeared.]*

Table 2-3. Operators

operator	arity	associativity	description
`++, --`	1	right	increment, decrement
`-, ~, !`	1	right	version, not
`*, /, %`	2	left	multiply, divide, modulo
`+, -`	2	left	add, subtract
`<<, >>`	2	left	arithmetic shift
`>>>`	2	left	logic shift right
`&`	2	left	bitwise and
`|`	2	left	bitwise or
`^`	2	left	bitwise exclusive-or
`<, <=, >=, >`	2	left	ordered comparison
`==, !=`	2	left	equality comparison
`&&`	2	left	boolean and
`||`	2	left	boolean or
`->`	2	left	implies
`?:`	3	right	if-then-else
`=, etc.`	2	right	assignment (see text)

The unary minus returns the negation of its numeric argument. Thus, evaluating any of the expressions `-1`, `-(1)`, and `-(2 - 1)` all return the value -1. Negation preserves the type of its argument: negating an int value results in another int value.

The bitwise inversion operator, `~`, returns, given an integer numeric argument, an integer numeric value of the same type, but with all 0 bits replaced by 1 bits, and vice versa. Thus, `~0` returns -1.

The boolean not operator, `!`, returns `FALSE` iff its operand has the default value for its type, and `TRUE` otherwise. For example, the default value of numeric types is 0, so `!0` will return `TRUE` and `!456` will return `FALSE`.

The value returned by `!` has the boolean type. The only values of this type are `TRUE` and `FALSE`, also known as `YES` and `NO`.

Apart from `>>>` and `->` the binary operators perform the same function as their C, and many other languages, equivalents. `>>>` shifts a signed number in the same way as `>>` shifts an unsigned number in C. The implication operator, `->`, performs the boolean implication: `a -> b` is equivalent to `!a || b`. Table 2-4 shows a few examples of their use.

Table 2-4. Examples of >>> and ->

expression	result
`128 >> 1`	`64`
`-1 >> 1`	`-1`
`-1 >>> 1`	`0x7fffffff`
`TRUE -> FALSE`	`FALSE`
`TRUE -> TRUE`	`TRUE`
`FALSE -> TRUE`	`TRUE`
`FALSE -> FALSE`	`TRUE`

The operators `*`, `/`, `+`, `-`, `<`, `<=`, `>=`, and `>` operate on any numeric type. The `%`, `<<`, `>>`, `>>>`, `&`, `|`, and `^` operate on integer numeric types. `&&`, `||`, and `->` operate on the boolean type. Furthermore, `==` and `!=` operate on any type, including those to be introduced later on.

All binary operators are left associative, except the assignment operators, `=`, `+=`, etc., which is right-associative. The order of evaluation of the operands follows the associativity of the operator. Thus, `1 + 2 + 3` is interpreted as `(1 + 2) + 3`, and the `3` is only evaluated after the addition of `1` and `2`---not that it makes any difference in this simple case. Furthermore, `a = b = c` means `a = (b = c)`. (We will return to the assignment operator later on---they aren't much use now when constants and operators are the only building blocks at hand.)

The boolean operators are *short-circuited*. This means that if enough information is known about the operands that the result of the whole expression is known the evaluation of any remaining operands is skipped. Thus, `FALSE && x` returns `FALSE` without ever evaluating `x`, since its value does not matter.

There is only one ternary operator, `? :`, also known as the if-then-else operator. To compute the maximum of `a` and `b`, one could use `a > b ? a : b`, meaning that if `a` is larger than `b`, the second expression (in this case `a`) will be evaluated and returned as the result; otherwise the result of the third expression (`b`) is the result of the whole expression.

In general, when used as `x ? y : z`, the type of `x` must be boolean, and the type of `y` must equal the type of `z`. Furthermore, only one of `y` and `z` will be evaluated.

Local variables

It is often necessary to keep the result of an expression for later use. A good example of this is a counter: if you count from 1 to 10, when you're ready to go to the counter's next value you must know what its current value is. A local variable can be used to hold such values. For example, if `a` is a variable, then the assignment

```
a = 1;
```

will assign to `a` the value `1`. When followed by the expression `a + 7`, the result of that expression will be `8`.

As you will have guessed, each variable has a type. Before a local variable can be used, it must be declared, not only to indicate its type but also to declare its existence. The variable `a` used above would be declared as:

```
int a;
```

You can experiment with variables by modifying the `main` method of our example program again.

```
int
  main Array argv
{
  int a = 11, b = 6;

  [[[stdio out] print ("a = ", a, " b = ", b)] nl];
  int c = a * b;
  [[[stdio out] print ("a * b = ", c)] nl];
  c = a + b;
  [[[stdio out] print ("a + b = ", c)] nl];

  return 0;
}
```

This example shows that multiple variables can be declared in one declaration, and that an assignment to a variable can be included in its declaration. In fact, when such an initial value is omitted, the variable will be set to `0`. Verify this by omitting the initialization of `b` (i.e., the '`= 6`') from the example and running the program again after rebuilding.

In general, if a variable declaration does not specify an initial value, the value of the variable will be the default value of the variable's type, e.g., `0` for numeric types and `FALSE` when the type of the variable is boolean.

Loops

One of the more valued features of computers is their ability to repeat, and repeat often. The *loop* is the crucial language construct underlying concept. Would you like to compute the conversion table from Celsius to Fahrenheit for every value between -100 and 100? Here's how to automate that.

```
int
  main Array argv
```

```
{
  float celsius = -100.0;

  while (celsius <= 100.0)
    {
      float fahrenheit = 32.0 + 9.0 / 5.0 * celsius;

      [[[stdio out] print (celsius, " ", fahrenheit)] nl];
      celsius = celsius + 1.0;
    }

  return 0;
}
```

The `while` loop tests its condition, in this case whether `celsius` still is less than `100.0`, and as long as the condition is true, the expression following it is executed. This expression is called the *body* of the loop. The loop condition is re-evaluated every time the body has been evaluated.

The example shows another noteworthy item: the body expression actually consists of a declaration and two expressions, separated by semicolons (`;`) and enclosed in braces, `{` and `}`. Such an expression sequence between braces is called a *compound expression*. The type and value of such a compound expression are the type and value of the last expression within the compound. This value is not always used, like in this example.

Two other loop constructs are available, which basically are nothing more than a variation on the `while` loop theme.

The first variation is the `do while` loop. For example, the following code prints the numbers from `0` to `9` (inclusive or exclusive?).

```
int
  main Array argv
{
  int counter;

  do
    {
      [[[stdio out] print counter] nl];
      counter = counter + 1;
    } while (counter < 10);

  return 0;
}
```

The difference with the plain `while` loop is that the condition is evaluated *after* the body has been executed, i.e., the body is executed always at least once, whereas the body of the `while` loop can be skipped if the loop condition evaluates to `FALSE` before the body would be entered for the first round.

The third, and last, loop variation is the `for` loop. Like the `while` and `do while` loops, its syntactical origins stem from C. Here is a Fahrenheit to Celsius converter for degrees between -100 and 100, written down using a `for` loop.

```
int
  main Array argv
{
  float f;

  for (f = -100.0; f <= 100.0; f++)
    [[[stdio out]
      print (f, " ", (f - 32.0) * 5.0 / 9.0)] nl];

  return 0;
}
```

The `for` loop starts with three expressions, separated by semicolons. The first expression, in this case `f = -100.0`, is always executed. The second expression is the condition of the loop. If it evaluates to `TRUE`, the body is evaluated, followed by the evaluation of the third expression, `f++`, which increments `f` by 1. Then the condition is re-evaluated, and, if it is `TRUE`, the body, and so on.

As has been stated at the start of this chapter, everything in a method is an expression. Obviously then, loops also have a value. The value of a loop is the value of the body expression the last time it was executed. If the body is never executed, as can be the case with `while` and `for` loops, the value returned is the default value for the type.

More operators

The last loop example showed an operator which was not yet introduced, namely `++`. Its effect is to increment its operand, `f` in the example, by `1`. Similarly, `f--` would decrease `f` by `1`.

The increment and decrement operators can be used in a postfix notation, as in the example, or a prefix notation, as in `--f`. The difference between these notations is the value returned by the expression. The postfix notation returns the value of the variable before the modification; in prefix notation, the value of the expression the new value of the variable.

Given the fact that the value of a compound expression is the value of the last expression contained in the compound, `f++` is identical to

```
{
  int g = f;
  f = f + 1;
  g;
}
```

and `--f` equals

```
{
  f = f - 1;
}
```

The last kind of operators to be introduced are the modifying assignment operators. For example, `f = f + x` can be written as `f += x`. The same is true for every other binary operator. The precedence of these operators is equal to that of the normal assignment (they are the ‘etc’ in Table 2-3).

Conditionals

Next to loops, conditional expressions are also important language constructs. The `?:` operator introduced in the section called *Operators* is an example of a conditional expression: *if* the condition is `TRUE`, *then* execute what follows the `?`, *else* execute what follows the `:`.

This section introduces a semantically equivalent to the `?:` operator, but, as with a lot of TOM language constructs, with a syntax pleasing the eye accustomed to C.

The following example shows the use of the `if else` construct.

```
int
  main Array argv
{
  int n = [argv length];

  if (n == 0)
    [[[stdio out] print "no arguments"] nl];
  else
    [[[stdio out] print (n, " arguments")] nl];
}
```

If this program is invoked without any arguments, the Array `argv` will not contain any elements, and will return `0` when asked for its `length`. Consequently, the program’s output will be `no arguments`. When invoked with at least one argument, it will report the number of elements in `argv`, which is the number of arguments.

When you have modified the `hello` program to have the above `main` method, the following is a example of its output when run.

```
$ ./hello
no arguments
$ ./hello tiny little program
3 arguments
$ ./hello "little robot"
1 arguments
```

And thus we discover a bug in our program: the output when invoked with a single argument is wrong since ‘1 arguments’ is not proper English. You are hereby challenged to fix this, armed with the knowledge that what follows an `else` is an expression, just like the `if else` is an expression.

In contrast to the `?:` operator, the `else` branch of an `if else` conditional is optional. If the condition evaluates to `FALSE` and the `else` branch is missing, the value of the whole expression becomes the default value of the expected type.

For example, after the following expressions, the value of `a` will be `0`.

```
int b = 3;
int a = if (b < 0) b;
```

Chapter 3. Methods

Now the basics of what can be put in a method have been explained we can move on to building new methods.

Definitions

As was shown by the `main` method in the previous examples, a method has a name, argument types, and a return type. The name can consist of more than a single part, in which case each part must be followed by an argument---remember how the `out` method of the `stdio` class does not need any arguments.

For instance, in the following example, the method's name is `multiply by`;

```
int
  multiply int a
        by int b
{
  return a * b;
}
```

This method accepts two arguments, each of type int, and returns another int. The body of the method simply returns the result of multiplying `a` and `b`.

Notice how every name part of the method starts on a new line, and that the nameparts are right justified. This is considered a readable style of writing method declarations. It is much more readable than written like this:

```
int multiply int a by int b { return a * b; }
```

To test our `multiply by` method we add it to the example program, between the start and the end of the class implementation, and test it using the following `main` method:

```
int
  main Array argv
{
  int i, n = [argv length], result = 1;

  for (i = 0; i < n; i++)
    {
      String s = [argv at i];
      result = [self multiply result by [s intValue]];
    }
  [[[stdio out] print result] nl];

  return 0;
}
```

The receiver of the `multiply by` message is `self`. `self` is an implicit argument to every method; it is the receiver of the message which caused invocation of the method. In this example, `self` is the `HelloWorld` class object.

This example also shows the use of a `String` object, which is retrieved from the array `argv`. An `Array` stores the objects it holds in a sequence, and using the `at` message, one can ask for the object at a certain index, as long as the index is within range, in this case 0 <= `index < [argv length]`.

The full declaration of the `at` method is

```
Any
  at int index;
```

That is, an `Any` object is returned and since we know it is a `String`, we are allowed to assign the value returned to a `String`. In fact, as the name `Any` suggests, we can assign an `Any` to a variable of *any* class.

In the example, the `String s` is asked for its `intValue`. `intValue` is a `String` method which returns the integer value held by the string, in a straightforward way. For example, when asked for its `intValue`, the string `"123abc"` will return 123, `"abc"` will return 0, and `"0x123abc"` will return 1194684.

It is customary for method name parts to use mixed case identifiers, starting with a lowercase letter. Class names also follow the mixed case convention, but they start with a capital letter. Variables should be all lower case, with the words within the identifier separated by '-' or '_'.

When recompiled, the `hello` program will, when run, output the result of multiplying its arguments.

```
$ ./hello 23 2
46
$ ./hello
1
$ ./hello 123456789 98
-786136566
```

The program behaves as expected, apart from the last example. Two positive numbers, when multiplied should return a positive number. However, when 123456789 is multiplied by 98 the answer, 12098765322, is too large to fit a signed 32 bits value, which is the int used for the `result` and the `multiply by` method. Obviously, the problem can be shifted to the 64 bits limit by using long values, but that is not a real solution; it is important that you realise the existence of such limits.

Tuples

There are occasions, when returning a single value from a method does not suffice, or when it is tedious to have multiple method name parts just to have multiple arguments. TOM solves both problems through tuples.

A *tuple* is a group of values within parentheses, separated by commas. For instance, `(1, 3.14)` is a tuple. The type of a tuple is the *tuple type* of which the elements are the types of the elements of the tuple. The type of the example is (int, float).

An example of a method using tuples is the `substring` method from the `String` class.

```
String
  substring (int, int) (start, length);
```

This method has a single argument, the tuple `(start, length)`. As with a lot of other `String` methods, this tuple is used to select a range of elements, starting at index `start`, and running for `length` elements, where a `length` of `-1` means infinity.

The following example program, when run, will show a running text, probably best viewed on a slow terminal (of, say, 2400 baud).

```
int
  main Array argv
{
  OutputStream out = [stdio out];
  String text = "Testing...  1 2 3", spaces = "          ";
  int len = [spaces length], finish = len + [text length];
  int num = (![argv length] ? 100 : [argv[0] intValue]);
  int i, count;

  for (count = 0; count < num; count++)
    for (i = 0; i < finish; i++)
      {
        if (i < len)
          {
            /* Case 1: Spaces followed by start of text.  */
            [out print [spaces substring (0, len - i)]];
            [out print [text substring (0, i)]];
          }
        else if (i < finish - len)
          {
            /* Case 2: Start and end the line in the text.  */
            [out print [text substring (i - len, len)]];
          }
        else
          {
            /* Case 3: Text followed by spaces.  */
            [out print [text substring (i - len, finish - i)]];
            [out print [spaces substring (0, len - (finish - i))]];
          }
        /* Go back to the start of the line.  */
        [out print '\r'];
        [out flushOutput];
      }
```

```
  = 0;
}
```

As an example of a method returning a tuple, let's look at using an `Enumerator`. An `Enumerator` can be used to traverse a collection of objects. After asking the collection for an `enumerator`, the method

```
(boolean, Any)
  next;
```

can be used to repeatedly retrieve the next object. The value returned by `next` is a tuple with a boolean and an object. If the boolean is `TRUE`, the second element of the tuple contains the object retrieved. Otherwise, if it is `FALSE`, the end of the collection has been reached.

The following program is an example of using this method.

```
int
  main Array argv
{
  Enumerator e = [argv enumerator];
  Any object;

  while ({
           boolean valid;
           (valid, object) = [e next];
           valid;
         })
    [[[stdio out] print ("got one: `", object, '\")] nl];

  return 0;
}
```

The example uses a compound expression as the condition of the `while` loop. This isn't just to show you this is possible; the tuple returned by `next` can not be used as a boolean condition, and the boolean must be repeated to give the compound its boolean value.

If the declaration of `valid` were not part of this compound (but put outside the `while` loop), the condition could be written as

```
while ({(valid, object) = [e next]; valid;})
  ...
```

which is the common notation.

In addition to providing multi-valued returns, and a compact method argument notation, tuples provide a sometimes desirable feature of simultaneous assignment. The most obvious application thereof is that of swapping the values of two variables:

```
(a, b) = (b, a);
```

Return values

Up to now, returning from a method was always written using `return`. This sets the value to be returned and terminates execution of the method. Sometimes it is desirable to set the return value but not immediately return from the method. Such constructions are common in languages which only provide the `return` statement, and resemble the following code snippet:

```
{
  ...
  int result = [self computation];
  [self deallocateResources];
  return result;
}
```

Using the *return assignment*, the return value of a method can be set, without causing immediate termination. A return assignment is written as an assignment with an empty left-hand side. The example then becomes:

```
{
  ...
  = [self computation];
  [self deallocateResources];
}
```

which is much cleaner. It is customary to only use `return` when immediate termination of the method is necessary. The return assignment is otherwise preferred.

When a method does not assign a return value, either through `return` or a return assignment, the value returned by the method will be the default value of the return type. Thus, the following two method definitions are equal:

```
boolean
  constantp
{
  boolean v;
  = v;
}

boolean
  constantp
{
}
```

When a method returns a tuple the return value returned must be set atomically, i.e. it is only possible to set all values in the tuple at once. There are situation where this is undesirable, and a solution to this is provided by *named return values*.

In the following example, the return type is followed by a tuple of identifiers `(valid, object)`. Each identifier denotes a local variable in the method, which is declared implicitly, and which is handled as a normal local variable. Upon return from the method, the value returned will be the tuple `(valid, object)`. Intermediate assignments to any of these variables will have the expected result of affecting the return value. Also, normal `return` and return assignment still operate as expected, i.e. they will affect the value of these variables.

```
(bool, Any) (valid, object)
  next
{
  valid = [self haveMoreObjects];
  if (valid)
    object = [self getNextObject];
}
```

If one of the return value names is the name of an argument, be it the implicit receiver object `self` or the message selector `cmd` or a normal argument, that part of the return value will correspond to the value of the argument. The creation of a local variable is omitted in this case. The following method for example simply returns the value passed to it:

```
int (value)
  echo int value
{
}
```

Note that the return value name is enclosed in parentheses, i.e. a singleton tuple. Return value names must always be a tuple, otherwise the compiler can't discern between the name and the return value or the first method name part.

A good reason to name the return values is that any documentation or comments about the method can simply refer to parts of the return value by name.

Chapter 4. Basic types

TOM has three kinds of types: basic types, objects and tuples. Furthermore, there is one special type, void, and there are two special type indications: dynamic indicates that the actual type will be dynamically checked; and id which, in denotes an *actual* object instead of the containing, declaring, *formal* object.

```
entity_type:
          basic_type
        | tuple_type
        | object_type
        ;
```

An `entity_type` is the possible type of an entity, such as an object variable, method argument or local variable.

```
argument_type:
          'dynamic'
        | entity_type
        ;
```

In addition to the usual types, an argument can have the dynamic type. The type actually passed will be encoded in the selector of the method being invoked. It is the responsibility of that method to retrieve the correct types as indicated by the selector.

```
return_type:
          'void'
        | argument_type
        ;
```

The type of value returned by a method can be anything that can be the type of an argument, plus void, indicating that the method will not return any value.

```
basic_type:
          'byte' | 'char' | 'int' | 'long' | 'float' | 'double'
        | 'boolean' | 'pointer' | 'selector'
        ;
```

Numeric types

TOM has two kinds of numeric types: integer and floating point. The integer numeric types are listed in Table 2-1. In places where a numeric type is expected, a narrower numeric type of the same kind is accepted and implicitly converted. Thus, a byte is acceptable as a char is acceptable as an int is acceptable as a long; and a float is acceptable as a double.

The default value for the numeric types is zero.

The boolean type

The boolean type is used for truthness values. It is extensively used in conditional constructs. The default value for the boolean type is falseness. The instance tom.All defines the following constants, each with the boolean type.

```
const TRUE = !0;
const FALSE = !1;
const YES = TRUE;
const NO = FALSE;
```

Therefore, any class which uses these constants must minimally inherit from instance tom.All or another class which does, such as `State`.

The pointer type

The pointer type is a rather abstract type. Values of the pointer type can not be operated upon; they can only be passed around; their primary use is for the implementation of arrays, integrating with foreign code, such as that written in C, and for debugging purposes. The default value for the pointer type is the invalid pointer (NULL in C). Note that, within TOM proper, it is impossible to refer to the constant null pointer.

The selector type

A selector is an abstract entity holding a name and typing information. A method invocation, also know as 'sending a message to an object', actually is the invocation of some behaviour identified by the object receiving the message, and the name of the message: the selector. The second part of a message are the arguments to the method. A selector is a name and typing information on the arguments carried by a message and return value expected by the sender of the message. Every method implementation has, as the second implicit argument the selector, `cmd`, used to invoke the method's behaviour.

The default selector value denotes the non-existing selector. The only operations defined on selector typed values are equality comparisons.

The void type

The void type is a special type: it indicates the absence of a value. Its most profound use is in typing the return value from methods; another use is as an expression where an expression is not needed but also not allowed. void is the only one void-typed value; no operations can be performed on void; and there is no default value for the void type.

The dynamic type

A method which is implemented outside TOM (say, in C) can have a dynamic return type and dynamic argument types. A dynamic type implies anything can be passed and will be accepted. The type of the value actually passed to the method is encoded in the selector, which the method receives as the implicit second argument (after `self`). Similarly, the method implementation can deduce the expected return type from actual selector. The dynamic type is used by, for instance, the `perform :` method, which is defined as

```
extern dynamic
  perform selector sel
        : Array arguments = nil;
```

If `perform :` is invoked as

```
int a, b, c;

(a, b, c) = [foo perform bar]
```

then it would be a (fatal) runtime error if the selector denoted by `bar` did not return a tuple of three integers.

Object type

Objects are the only way in TOM to create an aggregate value---there is no such thing as a struct.

- All objects---classes and instances---share the same *type*. Obviously, the compiler can tell a difference between a `Number` and a `ByteArray`, so not all objects are of the same *kind*.
- A variable with an object type actually is a reference to an object. All objects are allocated from a heap.
- In the rest of this document, often the type of an expression or entity is referred to. In most cases, this means the type of the value, or the kind of object if the type is the object reference type.

The default value of an object-typed variable is the invalid reference, `nil`. The type of `nil` is _builtin_.Any. Messaging `nil` results in the condition `nil-receiver` being raised.

```
object_type:
          class_name
        | `id'
        | `class' `(' object_type `)'
        | `instance' `(' object_type `)'
        ;
```

The plain `object_type` is a `class_name`; this indicates the *instance* of the indicated class. id indicates the type of the actual receiver. The variation with a 'class' or 'instance' shifts the meta level into the specified direction. Examples:

`Foo`

A reference to an instance of `Foo` (or an instance of a subclass of `Foo`).

`class (Foo)`

A reference to the `Foo` class object (or the class object of a subclass of `Foo`).

`instance (id)`

A reference to an instance of the current receiver. Obviously, it is an error if the current receiver itself is an instance, since instances do not have instances.

`class (id)`

A reference to the class object of the current receiver. For the current receiver being an instance, it denotes the class object; for a class it denotes the meta class object. For meta class objects, it denotes the meta class object of the `State` class.

As the receiver of a method invocation, a `class_name` denotes the class object.

The id type

The id type is not an actual type. In the context of an object definition, id is identical to the current class or instance being defined, in contrast with a normal type that would indicate the class containing the declaration; in the context of an invocation of a method involving id typed arguments or return values, id denotes the actual receiver of the method.

For example, if a class `Foo` declares the following method:

```
id self;
```

then the type of the expression

```
{
  Foo a;
  [a self];
}
```

is `Foo`. If there exists a class `Bar` which inherits from `Foo`, then the type of the expression

```
{
  Bar a;
  [a self];
}
```

is `Bar`, since it is a `Bar` that was the actual receiver of the `self` message (as far as the compiler knows).

The meta level of id can be shifted towards instances or classes, as can be seen in the following two method declarations from the `State` class.

```
<doc> Return a newly allocated instance
    of the receiving class.  </doc>
instance (id)
  alloc;

<doc> Return the class of the receiving object.  </doc>
class (id)
  class;
```

Thus, irrespective of the class of which the `alloc` method is invoked, the compiler knows that the object returned by `alloc` is an instance of the receiving class.

Similarly, the `isa` instance variable is declared by the `State` instance as

```
class (id) isa;
```

Thus, in the context of a subclass of `State`, the `isa` has the type of the actual class object, not just the `State` class.

Tuple types

A tuple is a hotch-botch of values. For example, `(123, 3.1415)` is a tuple, and its type is (int, float).

The tuple type is not a first-class type: it is impossible to declare a variable with a tuple type. A single element tuple type actually is the type of the element. The default value of a tuple type is a tuple with as elements the default values of the tuple type's elements. Tuples can nest: `((1, 2), (3, 4))` is a valid tuple, the type of which is ((int, int), (int, int)). Tuple nesting is rarely used.

The dynamic type can not be an element of a tuple type. All other types are allowed.

The primary use of tuple types is in passing values to or from a method. The following example declares a method `divmod` which accepts one argument being a tuple of two integers and which returns another tuple of two integers:

```
<doc> Return (a / b, a % b).  </doc>
(int, int)
  divmod (int, int) (a, b);
```

Another use of tuples is in shorthands such as simultaneous assignments:

```
int a, b; ...;
(a, b) = (b, a)
```

The evaluation of tuple elements is defined to be from left to right. Thus, the result of the following expression

```
{
  int i = 0;
  (++i, ++i);
}
```

is defined and equal to `(1, 2)`.

The type of an element of a tuple can be neither dynamic nor void.

Chapter 5. Classes

We now come to the most important subject of an object oriented programming language: how objects are constructed. We will use a simple `Counter` class as the example along which to explain the object basics. We know the following about `Counter` objects:

1. each `Counter` object maintains information about its current value, and
2. it responds to the `nextValue` message by returning the next value, thereby updating its current value.

This description only contains statements about the `Counter` objects, i.e. instances of the `Counter` class. Therefore the class definition can be empty, for now.

```
implementation class Counter end;
```

The `Counter` instances maintain a current value, and respond to the `nextValue` method. This is written thus:

```
implementation instance Counter
{
  int current_value;
}

int
  nextValue
{
  current_value += 1;
  = current_value;
}

end;
```

The definition of the `Counter` instance starts with `implementation instance Counter` and ends with `end;`. Within this definition, between the first pair of braces an *instance variable* is defined. Instance variables form the state maintained by each instance of the current class: every `Counter` instance will have its own value of the `current_value`. This variable has the type int.

Following the instance variable declaration, the `nextValue` method is defined. This method has no arguments and returns an int. In the body of the method, the `current_value` instance variable of the *current instance* is incremented by 1, and the resulting value is returned. The current instance is the receiver of the message as a result of which the current method was invoked. Thus, in the following example

```
Counter count = ...;
int i = [count nextValue];
```

the `count` variable denotes a `Counter` instance; in the second line, the `nextValue` message is sent to it, and the `nextValue` method will be invoked, with the object we know as `count` as the receiver.

Inheritance

We now have a class definition, but no means yet to create instances. The language does not provide a mechanism to create instances; instead, this functionality is provided by the library. The TOM standard library contains a class `State` that provides the following method:

```
instance (id)
  alloc;
```

This class method returns a new instance of the receiving class. Thus, invoking

```
x = [State alloc];
```

will return a new instance of the `State` class. “How does this help us to create `Counter` objects?” you ask. By indicating that a `Counter` object is also a `State` object, which will cause the `Counter` objects to behave as `State` objects, and similar, that the `Counter` class behaves as the `State` class. This is called *inheritance*: every method defined by `State` is not only applicable to `State`, but also to `Counter`. To indicate that our `Counter` inherits from `State`, the class definition is changed to:

```
implementation class
Counter: State

end;
```

Now any method or instance variable defined for the `State` class (or instance) now also applies to the `Counter` class (or instance). To denote the relationship between the two classes, `State` is called a *superclass* of `Counter`, and `Counter` is a *subclass* of `State`.

The methods inherited from `State` include the `alloc` method and we can now send `alloc` messages to the `Counter` class to create new `Counter` instances:

```
Counter count1 = [Counter alloc];
```

The return type of the `alloc` method is instance (id). The id type denotes the type of the receiving object, as seen by the caller. In the example, the receiver is the `Counter` class object, and id denotes the `Counter` class. The instance () modifies the type between the parentheses to indicate the instance of that type. The type of object returned by the `alloc` method in this example thus is ‘instance of the `Counter` class’, written as `Counter`, which is exactly the type of the `count1` variable to which the value returned is assigned.

Object variables

Classes and instances must be able to hold state. TOM provides a few kinds of state.

```
implementation class StatefulCounter
{
  int starting_value;
}

end;
```

As with the `Counter` instance defined previously, we've defined a `StatefulCounter`, which possesses state. In this case though, we've created a class variable, which maintains a single value, accessible by all instances of that class. However, each sub-class of `StatefulCounter` would receive its own copy of `starting_value` that would be shared by its instances.

Qualifiers

Various qualifiers may be applied to the declaration of an object variable. These qualifiers alter the behavior and scope of that variable.

mutable

```
mutable int starting_value;
```

A `mutable` variable automatically defines a setter method of the form:

```
void
  set_starting_value int _new_value
{
  starting_value = _new_value;
}
```

The compiler will generate the method using the correct type and variable names. This may be used in conjunction with the `public` qualifier.

obsolete

```
obsolete int starting_value;
```

An `obsolete` variable has been marked to warn programmers that it will be going away in the future. References to this variable will generate warnings. Additionally, if it is `public` or `mutable`, the methods generated will also be flagged as `obsolete` and use of those methods will generate warnings.

* *This does not completely work yet. Variable references do not yet generate warnings.*

private

```
private int starting_value;
```

A `private` variable is only accessible to methods on the class which defined it.

protected

```
protected int starting_value;
```

A `protected` variable is only accessible to methods on the class which defined it, as well as any sub-classes.

public

```
public int starting_value;
```

The compiler automatically creates an accessor method for a `public` variable. The accessor is of this form:

```
int
  starting_value
{
  = starting_value;
}
```

The compiler will generate the method using the correct type and variable names. This may be used in conjunction with the `public` qualifier.

static

```
static int starting_value;
```

A `static` variable only exists upon the class which defined it. Sub-classes do not receive their own copy of the variable. The `static` qualifier is only valid on class variables.

local

```
static local int starting_value;
```

A `local` variable's value is stored within thread-local storage. The `local` qualifier is only valid on static class variables.

redeclare

```
redeclare long starting_value;
```

Redeclaring a variable allows the programmer to change the type. Redeclaring a variable that was not previously declared is not allowed.

Method overriding

Suppose we need a `TwoCounter` object that increments its value by 2 instead of 1 each time `nextValue` is invoked. We could write a `TwoCounter` class from scratch en end up with a class very similar to `Counter`, but that wastes the effort we put in developing the `Counter` class and duplicates that effort in creating the `TwoCounter` class. The severity of this problem increases with the complexity of the classes and the effort needed to develop them, and to debug, test, document, etc.

Luckily, just like `Counter` inherits from `State`, `TwoCounter` can inherit from `Counter`. Since we need different behaviour for the `nextValue` message, the `TwoCounter` can provide its own `nextValue` method, overriding the method provided by `Counter`.

```
implementation class TwoCounter: Counter end;

implementation instance
TwoCounter

redefine int
  nextValue
{
  current_value += 2;
  = current_value;
}

end;
```

When `nextValue` is sent to a `TwoCounter` instance, the object will add 2 to its `current_value` and return the result. Apart from this method, a `TwoCounter` behaves exactly the same as a `Counter`.

Messaging `super`

In the previous section, the `nextValue` method in the `TwoCounter` class was a complete rewrite of the original method. If the method being overridden performs quite a heavy task, it is a waste to have to fully rewrite or copy it, when all that is needed is a slight modification before or afterwards. In the example, all that is needed by the nextValue of TwoCounter is an increase of the current_value before the code of the original method.

The original method can still be invoked, by messaging the special receiver super, as is show in this example:

```
redefine int
  nextValue
{
  current_value++;
  = [super nextValue];
}
```

The effect of messaging super is that the method invoked will be the method defined or inherited by the superclass of the current class. The value of self within that method will be the same as in the current method, thus messaging super is like messaging self, with the only difference being an indication which class is to provide the method implementation.

Messaging super need not just concern the method overridden by the current method. Any method can be invoked, though that is highly unusual, except in the case of initializers, where the designated initializer of the subclass invokes the designated initializer of the superclass. Though actually different these methods are identical *conceptually*.

The redefine qualifier shown in the example method definitions in this section are actually optional. The compiler can be directed to issue a warning when a redefine is omitted, but due to reasons to become clear later, the presence of redefine can not be required.

Object allocation and initialization

An object returned by alloc is in a known state: each instance variable of the object has the default value of its type (except the instance variables introduced by State, most notably isa, which is a reference to the object's class). The state of default values however is not necessarily a meaningful state. For example, if the specification of the Counter objects included that the first value normally returned by an instance is 10, the proper initial value of the current_value would be 9. Therefore, every object must be initialized after allocation. The default initialization method has no arguments; it is defined by State as

```
id
  init
{
  = self;
}
```

The conventional way of creating a new instance of a class is by invoking alloc and init in a single expression.

```
MyClass x = [[MyClass alloc] init];
```

If `Counter` objects would indeed return 10 as the first value returned from `nextValue`, the initialization method would look like this:

```
id
  init
{
  next_value = 9;
  = [super init];
}
```

Often, initialization of an object needs one or more arguments. An example of this is an initializer for the `Counter` class where the first value returned by `nextValue` can be specified.

```
id
  initWithValue int value
{
  next_value = value - 1;
  = [super init];
}
```

In case of multiple initializers, usually, one is the designated initializer and the other initializers can be implemented by invoking it. For example, if `initWithValue` were the designated initializer of a `Counter`, the `init` method could be implemented like this:

```
id
  init
{
  = [self initWithValue 10];
}
```

An advantage of this setup is that subclasses only need to override a single initializer, when necessary, instead of all or any number of them.

Always having to invoke two methods (`alloc` and some initializer) just to create a new object can become a burden. For this reason, classes can provide, through inheritance or by implementation, one or more *allocators*, which pack the allocation and initialization into a single method. For example, `State` provides `new` as the default allocator: `new` allocates a new object and invokes the default initializer.

```
instance (id)
  new
{
  = [[self alloc] init];
}
```

Using the allocator, objects can be created easier, though not faster since it involves an extra method call.

```
MyClass obj = [MyClass new];
```

Just like a class can have a different designated initializer than its superclass, it can also have a different designated allocator. The `Number` class for instance, provides an allocator `with` for easy allocation:

```
Number one = [IntNumber with 1];
```

Object destruction

The lifetime of objects is controlled by the garbage collector within the TOM Runtime. When no remaining strong references to an object remain, the GC will collect the object, deallocating the memory consumed by the object. Just before the GC collects the object, it invokes a deallocation notification method on the object whose default implementation is provided by `State`:

```
void
  dealloc
{
  void;
}
```

An object must override this method to perform cleanup operations specific to the needs and implementation of that object, such as closing files, shutting down database connections, or freeing OS resources. An example of this is `Descriptor`:

```
void
  dealloc
{
  if (descriptor != -1)
    bind ((stream-error, nil))
      [self close];
}
```

This method ensures that the `Descriptor` is closed, avoiding the leak of a file descriptor at the OS level.

When implementating a `dealloc` method, care must be taken to avoid messaging any other objects from within this method, as they may have become garbage as well, and already been collected. Since class objects can not become garbage, it is safe to message class objects.

There are no guarantees that an object will be collected in a timely manner, or even at all (should references remain). This makes it critical that garbage collected objects not be relied upon to manage the lifetime of very limited resources such as file descriptors. They should also not be used to manage objects whose collection is time critical, such as objects which are user-visible.

Polymorphism

A subclass understands the same messages as its superclass since every method defined by the superclass is inherited by the subclass if it doesn't provide its own method definition for the particular message. In the case of `Counter` and `TwoCounter` this means that any place where an object is handled as if it is a `Counter` instance, a `TwoCounter` instance can be substituted.

For example, the following method retrieves the `nextValue` from a `Counter` object passed as the argument:

```
int
  getNextValueFrom Counter counter
{
  = [counter nextValue];
}
```

When this method is passed a `TwoCounter` object instead of the expected `Counter`, sending the `nextValue` method is still valid. In fact, sending *any* message understood by a `Counter` will be valid, since the `TwoCounter` is a subclass of the expected `Counter` class. This observation is universally applicable: any time a certain class is expected, passing a subclass is equally valid.

Now, what happens when the following code is executed?

```
Counter c2 = [TwoCounter alloc];
int i = [c2 nextValue];
```

The variable `c2` has type `Counter`, but actually references a `TwoCounter` object. When the `nextValue` message is sent, which method is actually invoked: the one defined by `Counter`, because that is the type of `c2`, or the one defined by `TwoCounter`, because that is the actual class of the receiver? The answers is that the actual class of the receiver, and not the caller's idea of the receiver's type, decides which method is invoked. This is called *polymorphism*: the method invoked depends only on the receiver.

When looking back you'll notice that at a lot of places in the preceding sections, the polymorphism was already used, without mentioning it. Yet, intuitively, the meaning was always obvious.

A class is not atomic

A class defines the behaviour of its instances, and the state that they carry in support of that behaviour. You can use this class as-is, or subclass it to provide more specialized behaviour. This is where the story ends with most object oriented programming languages, but not with TOM. In circumstances where you have no control over a class, for instance because it is part of a vendor-supplied library, not being able to amend it to your needs might just mean that you can not use the class or, even worse, the whole library.

TOM classes are not atomic. A class can be modified: you can add methods, instances variables and superclasses, without modification to the original source, which is essential in case you do not have access to it. Modification can take place at compile, link, and run-time. Methods can also be replaced, for example to fix erroneous behaviour of the original implementation.

* *[A limitation of the current runtime library is that adding instance variables at run time to a class of which instances have been allocated is not possible.]*

Suppose the `Counter` class we have used as an example throughout this chapter is supplied to us without the source code. It doesn't fully suit our needs, but we do not want to waste any effort rewriting it and we have no control over some locations in the code where `Counter` instances are allocated, making subclassing not an option. We can write an *extension* of `Counter` to make it suitable for our purpose.

Suppose the needed extra functionality is a `current_value` method which returns the current value of the `Counter`. We can supply this functionality in the following extension:

```
implementation instance
Counter extension fix

int
  current_value
{
  = current_value;
}

end;
```

Multiple inheritance

Multiple inheritance refers to the ability of a class to have more than a single superclass. Various object-oriented languages provide multiple inheritance; equally many languages provide only single inheritance, possibly with 'interface inheritance' constructs like Objective-C's protocols and Java's interfaces. C++ at one point in its committee life, in all its baroqueness, had both multiple inheritance, plus inheritance of interface through signatures.

Semantics

Suppose the class `D` has both `B` and `C` as a superclass.

```
implementation class D: B, C
end;

implementation instance D
end;
```

What effect does this inheritance of two classes have?

- Any state defined for instances of `B` or `C` is also present in instances of `D`. There is no sharing of slots based on the name of instance variables as in CLOS. Thus, every instance variable `i` consuming space in an instance of `B` or `C`, also consumes space in an instance of `D`.
- Every method defined for `B` is also defined for `D`. Obviously, every method defined for `C` is also defined for `D`.
- If both `B` and `C` define the same method `foo`, a method clash is said to have occured, and `D` should provide its own implementation of that method. This is not mandatory; it is not checked by the compiler; it is optionally checked by the resolver. If a method clash is not resolved, a `program-condition` is raised when the method is invoked at run time.

A class with instances that carry state (i.e., instance variables) must be a subclass of the `State` class. If both `B` and `C` maintain state, they must both inherit from `State`, which brings up the issues involving repeated inheritance. Actually, these issues are mild in TOM when compared to the same issues in languages like C++ or Eiffel. To sum it up: repeated inheritance is shared inheritance.

- With respect to instance variables, things remain the same: every instance variable declared in a superclass gets its spot in the subclass. Thus, `D` only has one `isa` instance variable (inherited from `State`), even though it 'is inherited twice'.
- If a method `foo` is defined by `State`, unharmed by `B`, and redefined by `C`, it is the redefinition of `C` that is applicable to `D`. The implementation by `State` that 'is visible' through the inheritance of `B` is nulled by it being overriden in the inheritance path through `C`.

Messaging super

Messaging `super` is performing an invocation of a method as provided by a superclass. Usually, the method is invoked from the method that overrides the original definition. For example, the following is not uncommon for an initializer:

```
id
  init
{
  my_counter = 1;
  = [super init];
}
```

If a class has multiple superclasses, the message to super must indicate which superclass is to provide the method implementation. If the super message is unambiguous, the compiler will make the obvious choice, that can be described as follows: Suppose class `D` overrides the method `init` as described above. Imagine a class `E`, with exactly the same superclasses as `D`, but without overriding any method. Then, invoking `init` on an instance of `E` will be bound to a particular method implementation. If that implementation is provided by a direct superclass of `E` or is only visible through a single direct superclass of `E`, than that is the superclass a message to super in `D` refers to. If, however, there is a method clash, or the method is visible through more than one direct superclass, the super reference is ambiguous and must be disambiguated, as shown in this example (note that the syntax of directing which super to message is different from 'casting super'):

```
id
  init
{
  my_counter = 1;
  = [super (B) init];
}
```

Net effect is that when dynamic loading introduces an `init` method for instances of class `C`, that change will not be applicable to this method and its messaging of super.

Special classes

Several classes can be recognized in any TOM program, much like the standard classes in other languages. For example, in Smalltalk, the `Object` class resides at the root of the inheritance tree. TOM employs the following special types c.q. classes:

`Top`

: The implicit supertype of all object types. This type is not very useful, as it does not define any behaviour and can not be extended.

`Any`

: The implicit subtype of every object type: when used as the return type of a method that can return any object, the caller never needs to cast the value that is returned. For example, the following method is the only object retrieval method of the `ObjectArray` class (which offers a read-only array abstraction that stores object references) that actually directly retrieves an object:

```
Any
  at int index;
```

`All`

: The conventional supertype of all object types. All classes *should* either inherit from `State` (see below) or both class object and instances *should* inherit from the `instance (All)`. The instance `All` defines all kinds of behaviour that is useful for *all* objects, both class objects and instances.

Being the supertype of all objects, `All` can be used as the type of a formal argument, allowing any object type to be passed as an actual argument, without needing a cast. This is used, for example, by the only method of the `MutableObjectArray` class (which offers a read-write array abstraction that stores object references) that actually directly modifies the array:

```
void
```

```
set All object
 at int index;
```

`State`

Every class must inherit from `State` for the instances to be allocatable. `State` is also the class that defines the `isa` object variables and that provides the designated way to create new instances, namely through the `alloc` class method.

The `instance (All)` is the conventional supertype of all objects, a fact that is visible in the definition of the `State` class:

```
implementation class State: instance (All)
...
end;

implementation instance State: instance (All)
...
end;
```

The usefulness of the `Top` and `Any` types is restricted to compile time: they do not represent real objects that can be allocated or extended. The pervasive presence of `All` enables the addition of behaviour to *all* objects, not discriminating between instances and class objects, simply by extending the `All` instance. Similarly, behaviour can be added to all classes *or* all instances, simply by extending the `State` class *or* the `State` instance.

The `State` is a superclass of all classes that specify allocatable instances. `State` specifies the object variables that are needed in every instance and class object. `State` also is the class that provides the object allocation mechanism, through the `alloc` class method.

Every instance is described by its class, and every class by its meta class. The `State` meta class is also the meta-meta class of every meta-class. To prevent cycles, `State` can not inherit from classes that define class methods.

Collections

More coming.

Chapter 6. Advanced topics

Blocks

This section discusses blocks as they were added to TOM by Tesla. The old TOM compiler, tomc, can't handle blocks.

Basics

A block is a piece of code that of which the execution is postponed. Instead of the code being executed immediately, an instance of the `Block` class is created as a placeholder for the piece of code. When the `Block` is evaluated, through its `eval` method, the piece of code is actually executed. Here is an example of our favourite program, using a `Block`.

```
int
  main Array arguments
{
  Block b = |{ /* no arguments */
            || [[[stdio out] print "hello, world"] nl];
            }|;
  [b eval];
}
```

A `Block` starts with |{ and ends with }|. In between are at least two vertial bars ('|') that precede the actual code that is contained in the `Block`. The verbose spacing, comment, and assignment to a variable are optional. The same program could look like this:

```
int
  main Array arguments
{
  [|{|| [[[stdio out] print "hello, world"] nl]; }| eval];
}
```

A `Block` can have arguments and it can return a value. The arguments are declared like the argument accompanying a method name part: either a single argument name preceded by its type, or an argument tuple preceded by a tuple type. The return type of a `Block` is not declared: the value returned by a `Block` is the value of the last expression and its type is deduced by the compiler. In the following example, we employ a `Block` to return the result of adding two integers:

```
int
  main Array arguments
{
  Block adder = |{ (int, int) (i, j)
                ||
                   i + j;
```

```
                |};
  int result = [adder eval (24, 42)];

  [[[stdio out] print ("24 + 42 = ", result)] nl];
}
```

A `Block` does not need to be evaluated lexically within the enclosing method; it can be evaluated from anywhere. Irrespective of the context in which it is *evaluated*, a `Block` can reference the variables in the context where it was *created*. In the following example, the variable `count` is incremented twice, and the number printed is 2.

```
void
  do Block a_block
{
  [a_block eval];
}

int
  main Array arguments
{
  int count;
  Block b = |{ || count++; }|;

  [self do b];
  [b eval];

  [[[stdio out] print ("count = ", count)] nl];
}
```

When a `Block` references variables from its context, the `Block` is invalidated when that context exits. The following example shows how such a situation may occur: the `Block`, created in the `one_block_please` method, is evaluated after the `one_block_please` method has exited. As a result, the `eval` method raises a `Condition`:

```
Block
  one_block_please
{
  int num_invocations;

  = |{ || ++num_invocations; }|;
}

int
  main Array arguments
{
  Block block = [self one_block_please];

  // FAIL: the context in which the BLOCK
  // was created has already exited.
```

```
  int n = [block eval];

  [[[stdio out] print ("number of invocations = ", n)] nl];
}
```

Apparently, a `Block` can not use variables from its context when that context exits before the useful life of the `Block` has passed. Since `self` is part of the context, also instance variables can not be used for that purpose. To remedy this, thus allowing a `Block` to maintain information over invocations irrespective of whether its context has exited, a `Block` can employ *block variables*, as shown in the following example, where the `Block` created has one block variable, `num_invocations`. Notice how block variables occupy the space between the double vertical bar we used up to now:

```
Block
  one_block_please
{
  = |{ /* no arguments */
     | int num_invocations;
     | ++num_invocations;
    }|;
}

int
  main Array arguments
{
  Block block = [self one_block_please];
  int n = [block eval];

  [[[stdio out] print ("number of invocations = ", n)] nl];
}
```

If you want to reference instance variables from a `Block` after its context has exited, you can do so by declaring a block variable `self`, as the following example shows. Note how the value that is assigned to the block variable `self` is the value of the implicit method argument `self`.

```
implementation instance
TOM
{
  int num_cows;
}

Block
  cow_counter
{
  = |{ /* no arguments */
     | id self = self;
     | num_cows++;
    }|;
}
```

```
int
  main Array arguments
{
  Block cc = [self cow_counter];
  int i;

  for (i = 0; i < 10; i++)
    {
      int cow_i = [cc eval];
      [[[stdio out] print ("cow number ", cow_i)] nl];
    }

  [[[stdio out] print (num_cows, " cows")] nl];
}

end;
```

The `eval` method

The `eval` method of the `Block` class is declared thus:

```
dynamic
  eval dynamic arguments;
```

The `eval` method accepts any argument and returns any type of value. The types of the actual argument and the value returned is deduced by the compiler and administered in its output. At run time, the `eval` method checks to see that the arguments passed match the arguments expected by the `Block`, and that the type of value returned by the `Block` matches the type of value that is expected by the caller. Upon a mismatch, a `Condition` is raised. As an exception, any type returned by a `Block` matches an expected return type void.

The overhead in execution time of the `eval` method is similar to that of the `perform with` of the `All` instance. This overhead is somewhat reduced by providing `eval` methods that are specialized on their types. As an example, this is what the `eval` method looks like that accepts an int and returns an int:

```
int (result)
  eval int a1
{
  pointer fn = code;

  if (check_block_selectors && cmd != arguments)
    if ([self arguments_fail (arguments, cmd)])
      return;

  <c>
     result = ((tom_int (*) (void *, void *, tom_int)) fn)
```

```
                                               (self, cmd, a1);
  </c>
}
```

In these specialized `eval` methods, the following variables and methods are used:

`code`

: an instance variable of the `Block` that points to the C function which is executed to evaluate the `Block`;

`check_block_selectors`

: a *class compile option* that is usually `TRUE`; and

`arguments`

: a selectors that describes the formal argument and return type of the receiving `Block`.

`arguments_fail (formal, actual)`

: a boolean method that returns `TRUE` if the `actual` selector passed to an `eval` method can not fit the `Block`'s `formal` selector. In fact, when `arguments_fail` is about to return `TRUE`, it will raise a `Condition`.

The `Block` class comes with some specialized `eval` methods, including this example `int eval int`. Additional specialized `eval` methods, can be added in an extension of the `Block` class.

* *TODO: Execution-speed measurements.*

Conditions

Conditions in TOM are modeled after Common Lisp conditions (see *Common Lisp the Language, 2nd edition* (http://www.cs.cmu.edu/Groups/AI/html/cltl/cltl2.html)), with some simplifications. Conditions are not like exceptions in languages like C++ or Java, the most important difference being that the stack is not unwound while the condition is being handled.

Creation and issuing of conditions is functionality of the TOM runtime library. Handling conditions is part of the TOM language. We'll start the discussion of these two intertwined subjects with catching things that are thrown.

Non-local gotos

TOM provides exactly one way to perform a non-local goto: *throwing* something at an object, by invoking the `throw` method that is defined by the `All` instance, as in:

```
[my_object throw void];
```

Here, void is not the void type, but the single possible value of that type, which bears the same name. Thus basically, we throw nothing special at the object we know as `my_object`.

Of course, throwing something is not really interesting if you can not catch it. To setup a catch for a value thrown at an object, use a `catch` expression:

```
catch (my_object)
  [foo do_difficult_with bar];
```

`catch` is followed by the *tag* expression in parentheses, followed by a body expression. While the body of the catch is evaluated, anything thrown at the tag will be caught by this catch. In the example, we're catching values for the object we know as `my_object`.

The tag expression must be an object. The tag can be `nil`, but that is not very useful, since you will not be able to throw anything at `nil`.

A `catch` is an expression and like every expression, it has a type and it produces a value after evaluation. The value of the catch expression is either the value of the body expression or the value thrown at the tag object. For example, the following is an elaborate way to assign to the boolean `a` whether `a1` is larger than `a2`.

```
boolean a = catch (self)
              ({
                 if (a1 > a2)
                   [self throw TRUE];
                 FALSE;
               });
```

When executing, if `a1 > a2`, the value `TRUE` will be thrown at the catch: execution of the body expression is terminated, the stack is unwound up to the stack frame of the catch (think `longjmp` if you like), and the value `TRUE` is returned as the value of the `catch` expression. In the other case, if `a1` is not larger than `a2`, the body of the `if` is not executed, and the result of the catch is the value of the last expression, `FALSE`.

Incidentally, note the parentheses around the body compound of `catch`: the body of a catch must be an expression, and a compound can only be an expression as an element of a tuple. (The tuple has only one element in this case.)

It is an error if the type of value thrown at an object does not match the type that the catch expects. The one exception to this rule is when the value void is thrown, in which case the value returned by the catch is the default value of the type to be returned (`0` for numbers, `nil` for objects, etc).

Issuing conditions

A condition is an object, an instance of the `Condition` class. A condition is normally created by invoking the following method of the `Condition` class:

```
instance (id)
```

```
    for All object
  class ConditionClass condition_class
message String message;
```

The three arguments provide the values of the three instance variables with the same name, of the `Condition` object that will be created and returned. The `object` is the object to which the condition applies, for example the `File` object for a condition applying to some failed operation on that file. The message is meant to provide a description of the condition, to be read by a human and not to be interpreted by a program. An example of a useful message is the string returned by the C library function `perror`.

The `condition_class` is an instance of the `ConditionClass` class. Each condition-class object has a name, and through their super-condition-class instance variable, the condition class objects describe a single inheritance hierarchy of condition classes.

The `Conditions` class contains static class variables for the predefined condition classes. For example, one of them is the `nil-receiver`; it is a `runtime-condition`, which in turn is a `serious-condition`, which is a `condition`. The `condition` is the root of the condition-class hierarchy; it is the supercondition-class of all other condition classes.

As an example, when a message is sent to `nil`, the following condition is created:

```
Condition c = [Condition for nil class nil-receiver
                         message "nil receiver"];
```

The object to which the condition applies is of course `nil`, since the fact that it is `nil` was the reason for creating the condition.

Conditions can be issued in two different ways: raised or signaled. When a condition is raised, as in

```
Condition c = ...;
[c raise];
```

execution of the program is guaranteed to not return from the `raise` method, exiting the program if that is the only way to achieve the goal. A condition is signaled by invoking its `signal` method:

```
Any
  signal;
```

An invocation of the `signal` method may or may not return, depending on the behavior of the installed condition handlers: if one of them performs a non-local goto, the `signal` invocation will not return.

Condition handlers

To start with an example, the following is a `main` method that I used frequently to observe unhandled condition signals (until the library option `:rt-signals` was provided that does the same):

```
int (retcode)
  main Array arguments
{
  ConditionClass cc = condition;

  bind ((cc,
         {
           [[[stdio err] print ("unhandled condition: ",
                                condition)] nl];
           condition;
         }))
    retcode = [self real_main arguments];
}
```

The `bind` sets up a condition handler, which will be in place while the body of the `bind` is active, in this case during the invocation of `real_main`. When a condition is raised or signaled, all active handlers are considered in reverse order from their creation.

Each handler in a `bind` is a two-expression tuple; multiple handlers are separated by semicolons. The type of a handler is

```
(ConditionClass, All)
```

The first element is a `ConditionClass`, indicating to which kind of conditions the handler applies. In the example, the condition class is `condition`, but a variable with a different name is used to denote it. The reason for this is the fact that within a handler, `condition` is the name of the condition that is being passed. In the example, a handler is installed that matches any condition with condition class `condition`, or any subcondition-class thereof.

The second element of a handler is an expression that will be executed to handle a condition. The condition is available in the implicit argument to the handler:

```
Condition condition;
```

A condition handler can do one of three things:

1. Let the condition pass: in this case the handler decides that the condition is not interesting after all, and that the condition must continue to search for a handler that is willing to handle it. Searching continues with outstanding handlers further on the stack.

   A handler shows its desinterest by returning the `condition` object, as is done in the above example.

2. Handle the condition: the handler returns any object, just not the `condition` being handled. What the value that is returned *means* depends on the kind of condition. In this case, signaling the condition will finish and the invocation of the `signal` of the condition will return with the value returned by the handler. See below for an example of this usage.
3. Perform a non-local goto, by throwing some value at some `catch` tag.

The above description is valid for a condition that is signaled. If a condition is raised, cases 1 and 2 are not discerned (and handled like case 1): if a condition is raised, some handler somewhere along the line must perform a non-local goto, or the program will exit when it has run out of possible handlers. A handler can ask a condition whether it is being signaled or raised by invoking its `raised` method, which returns a boolean.

`unwind`

The `unwind` expression guarantees that some protection code is executed, either when the body expression has been evaluated, or when the stack frame is cleared because of a non-local goto. In the following example, the `unwind` ensures that the `file` is closed no matter what happens:

```
File file = [File open ...];
unwind ([file close])
  ({
    /* Do something with the FILE.  */
    ...;
  });
```

The value of the `unwind` expression is the value of the body.

`signal` example

This section presents an example in which conditions are used to give the user control over what happens when a file can not be opened. This setup is only possible because the stack is not unwound while a condition is handled, i.e., because conditions can not only be raised but also signaled.

Suppose a (fictuous) `ReadOnlyFile` object is opened and created using this method:

```
ReadOnlyFile (the_file)
  open String filename
{
  the_file = [[self alloc] initWithFilename filename];
  if (![the_file open])
    return nil;
}
```

Thus, opening a file first creates a new object for the given filename, and then lets the file open itself. The implementation of the `open` method could look like this; hopefully the methods being invoked have descriptive enough names for their missing implementation to not be a problem:

```
boolean (success)
  open
{
  for (;;)
    {
```

```
      /* Try to open, and return TRUE upon success.  */
      [self attempt_to_open];
      if ([self is_open])
        return TRUE;

      /* See if we can get an alternative filename.  */
      Condition c = [Condition for self class file-open-problem
                               message [self strerror]];
      String alternative_name = [c signal];
      if (!alternative_name)
        {
          /* The condition was not handled.  */
          return FALSE;
        }

      /* Make the ALTERNATIVE_NAME our new name.  */
      [self set_name alternative_name];
    }
}
```

This method will try to open the file, getting alternative names as long as they are supplied, and finally return `TRUE` upon success, or `FALSE` upon failure.

We could use the mechanism provided by the `ReadOnlyFile` class in a program to offer the user the possibility of specifying an alternative filename, as shown in the following code:

```
/* Setup a handler for FILE-OPEN-PROBLEM conditions.  */
bind ((file-open-problem,
         {
           /* Retrieve the file object to which the condition applies.  */
           ReadOnlyFile file = [condition object];

           /* Describe the problem to the user and prompt for input.  */
           [[[stdio err] print ("trouble opening: ", [file name])] nl];
           [[[stdio err] print ("reason: ", [condition message])] nl];
           [[[stdio err] print "alternative (RET for original)? "] nl];

           /* Read a line of input.  */
           String input = [[stdio err] readLine];

           /* If the input is an empty line, return NIL, indicating that
              we did handle this condition, but that an alternative
              filename was not entered.  Otherwise, return the filename
              entered by the user.  As long as we do not return the
              CONDITION object, we will have handled this condition.  */
           [input length] > 0 ? input : nil;
         }))
  ({
    file = [ReadOnlyFile open "/foo/bar"];
```

```
  });
```

Below are two example runs; the program attempts to open the file and prints the value that is returned before exiting:

```
$ ./rofile foo
trouble opening: foo
reason: No such file or directory
alternative (RET for original)?
bar
trouble opening: bar
reason: No such file or directory
alternative (RET for original)?

*nil*
$ ./rofile foo
trouble opening: foo
reason: No such file or directory
alternative (RET for original)?
rofile
#<ReadOnlyFile 00201700 name=rofile>
$
```

Glueing TOM and C

At the lower levels of abstraction, it is often necessary to glue code written in one programming language to code written in another. Most languages can interface to C and, in fact, the TOM compiler translates TOM code to C code. Therefore, TOM provides extensive support for interaction between TOM code and C code.

C functions for TOM methods

There are two ways of mixing C code with a TOM program. One is straightforward and could be called elegant, certainly with respect to the other one, which is a hack. The straightforward mix of TOM code and C code is by implementing TOM methods in C. To inform the compiler of this setup, the method is qualified `extern`.

```
extern double
  cos double arg;
```

To a TOM compiler, this declaration doubles as a definition: a method declared `extern` can not have a method body. Though the actual C (or other language) function implementing this method is beyond the control of tesla or tomc, it is mandatory that the function is provided, or the resulting program will not link.

To implement the `cos` method in C, we need to know a little more about the name of a C function that implements a given method. In general, the C function name of a method has the following structure:

```
ic_unit_extension-name_mangled-selector
```

where each element has the following meaning:

`ic`

This is `i` for an instance method; `c` for a class method.

`unit`

The name of the unit containing this method definition. If the method is defined in a class, it is the unit containing the class. If the method is defined in an extension, it is the unit containing the extension, which is not necessarily equal to the unit containing the class.

For example, the `too` unit defines a `Proxy` extension to the `State` class, which itself is defined in the `tom` unit. For methods defined in this extension, the `unit` element will be `too`, not `tom`.

`extension-name`

This is the class name for a method defined in a class, i.e., in the *main extension*, or the composite name `Foo_Bar` for the `Bar` extension of the `Foo` class.

`mangled-selector`

This is the mangled selector name, i.e., the name of the selector after it has been mangled to fit the restrictions imposed on a C identifier: all characters that are not allowed in such an identifier are replaced by an underscore '_'. Given the kinds of characters that can occur in a selector name, this means that every '(', ')', '-', or ':' is replaced by an '_'.

Selector names

Before we can continue implementing the `extern` method, we need to know how the name of a selector is constructed. This is best explained starting with the method that is invoked when a message with that selector is sent to an object. Suppose the `cos` method is invoked, then the selector contains its name, `cos`, and an encoding of its return type and argument type. Table 6-1 lists all TOM types and for each type the character that is used to encode that type.

Table 6-1. type encodings

type	encoding	type	encoding	type	encoding
void	`v`	int	`i`	pointer	`p`
boolean	`o`	long	`l`	selector	`s`
byte	`b`	float	`f`	*reference*	`r`

type	encoding	type	encoding	type	encoding
char	`c`	double	`d`	dynamic	`x`

The selector name is made up of method names and encoded argument types, preceded by the encoded return type. Each type is enclosed in parenthesis. The selector name of our double returning `cos` method accepting a double argument becomes:

```
(d)cos(d)
```

A few words need to be said about the reference and dynamic types in Table 6-1. First, a *reference* is not a TOM type: there is not a type in the language that has the concrete syntactic representation `reference`. A reference stands for a reference to an object, *any* object. Thus, at the level of selectors and selector names, all objects are equal.

The encoding of the dynamic type only occurs in method names (in the mangled selector part), never in the selector of a message. For example, the name of the function implementing the instance method

```
void
  print dynamic a;
```

of the `Bar` class in the `foo` unit will be

```
i_foo_Bar_v_print_x_
```

but when a message is sent that will invoke this method, the actual arguments of the message are known, and the selector passed to the method will convey their types. Thus, when invoking the method like this:

```
foo.Bar mybar = ...;
[mybar print FALSE];
```

the selector that is passed to the method will be `(v)print(o)`, showing that for the dynamic formal argument, the actual argument passed is a single boolean.

As an example of the encoding of a tuple, for the following invocation of `print`

```
[mybar print (3.14e0, 9876543210, FALSE, 1234567890, 1.6d-19)];
```

the selector passed to the method will be

```
(v)print(floid)
```

Tuples can of course also occur in a method name, and hence, in mangled form, in the name of a C function implementing that method. The following method

```
extern double
  atan2 (double, double) (x, y);
```

responds to the selector `(d)atan2(dd)`, and the C function implementing this method in the `Math` class of the `C` library unit is `c_C_Math_d_atan2_dd_`.

Type names

Before we can implement our `cos` method, we must know how to denote the TOM types in C. Table 6-2 lists the TOM types and for each type the equivalent type to be used in C. These types are defined in `<tom/trt.h>`.

Table 6-2. C types for TOM types

TOM type	C type	TOM type	C type	TOM type	C type
void	void	int	tom_int	pointer	void *
boolean	tom_byte	long	tom_long	selector	selector
byte	tom_byte	float	tom_float	*reference*	tom_object
char	tom_char	double	tom_double	dynamic	`...`

The triple dots as the C type for the TOM dynamic type actually refer to the triple dots used in a C function to denote a variable number of arguments and the used of `<stdarg.h>`. However, that is a very hairy issue we will not delve into right now.

External implementation

From the information in Table 6-2, we are finally able to write our `cos` method, supposedly for the `Math` class of the `C` unit:

```
#include <math.h>
#include <C-r.h>

tom_double
c_C_Math_d_cos_d_ (tom_object self, selector cmd, tom_double arg)
{
  return cos (arg);
}
```

The conversion from the tom_double arg to the (C) double accepted by the function `cos` is handled by the C compiler, as is the conversion of the result of `cos` to the value that is returned. (On all machines currently supporting TOM, a tom_double is simply a double, making the conversion rather easy.)

A few things can be said about this code:

- The inclusion of `<C-r.h>` isn't strictly necessary in this example, but in the case of less trivial implementations, including the file *unit*`-r.h` is mandatory. This file is the resolver output and contains vital information about the classes and selectors that are defined in the *unit*. It also includes the resolver output of the units on which the *unit* depends, plus the TOM runtime header file `<tom/trt.h>`. The latter is mandatory (for the C equivalent definitions of the TOM types). Often you will also find use for including `<tom/util.h>` which contains less elementary information for interfacing with TOM code and the TOM Run Time library (trt).

- The first argument to a method implementation is always the (implicit) receiver object. In C you should always declare the type to be a tom_object, even if you think to know that it will be something more specific. The tom_object type is pretty opaque, being defined as follows (in `<tom/trt.h>`:

  ```
  typedef struct trt_instance
  {
    /* The class of this object.  */
    struct trt_class *isa;

    /* The flags needed by the runtime.  */
    tom_int asi;
  } *tom_object;
  ```

  The `isa` is the pointer to the class of the object; the `asi` field is used by trt to store (1) whether the object is an instance, a class, or a meta class, and (2) information for the garbage collector. Any instance (or non-static class) variables of the object are not directly available by dereferencing `self`; a future TOM highlight will shed light on how that can be achieved.

- The selector argument `cmd` is the second implicit argument to every method invocation. The `arg` is the first 'real' argument.

The hack

As promised at the start of this highlight, there also is a hack to write functionality in C. This hack uses the fact that the output of Tesla actually is C.

Normally, the TOM compiler ignores anything enclosed within `<foo>` and `</foo>`, regarding it as comment (which does not nest). The flexibility of this commenting scheme is that special comments, i.e., 'comments' with more meaning than just some remark on the code to follow, can be qualified. For example the TOM documentation generator extracts comments enclosed in `<doc>` and `</doc>`, regarding them as documentation on the class, variable, or method to follow. It skips all other comments, including the `<copyright>` and `</copyright>` at the top of the TOM library units source files, since copyright information is not interesting for the reader that wants to learn how a certain class works.

The single exception to the above rule is that text enclosed with `<c>` and `</c>` is *not* taken to be comment. Instead, the enclosed text is copied verbatim to the output, implying that the text better be literal C code, which is actually what it was meant for.

Our `cos` method can now be written as follows:

```
<c>
#include <math.h>
</c>

<doc> Return the cosine of the argument {arg}.  </doc>
double (result)
  cos double arg
```

```
{
<c>
  result = cos (arg);
</c>
}
```

Again, a few notes:

- Do not use any nasties like the C `return` statement in your C code. Instead, assign a value to the return value of the method, as is done in this example.
- You can include header files like `<math.h>` (at the global level; not within a method) but you can not include the resolver output as was done with the external implementation of `cos`. This has some implications that increase the complexity of including C code like this.
- If you C code starts with a declaration, it should start its own block.
- The C code is included literally, so it does not need to be a fully delimited entity. For example, the following implementation of `cos` is ‘legal’:

```
double (result)
  cos double arg
{
<c>
  {
    double a = arg;
    result = cos (arg);
</c>
    return;
<c>
  }
</c>
}
```

Interaction with the Garbage Collector

...

Method forwarding

Forwarding mechanism

One of the things that make dynamic binding an interesting approach to method dispatching (i.e., the decision which code to invoke to handle a given message) is the ability to forward a method invocation. A message is forwarded when the receiver object does not provide an implementation of the method denoted by the message. In this highlight, the forward mechanism of TOM will be explained.

Every method has two implicit arguments: `self` and `cmd`. The object `self` is the 'current object'; it is the receiver of the message that lead to the method invocation. The argument `cmd` is the selector of that message.

When an object does not implement a method, the `self` and `cmd` arguments are used to decide how to respond to the message.

- The receiver is asked for its `forwardDelegate`. If an object other than `nil` is returned, the message is simply resent to that object. This is the mechanism of choice when a lot of methods are to be delegated to few different objects.

  The `forwardDelegate` method is implemented by the instance All, and thus (by convention) implemented by all classes and instances. The default implementation simply returns `nil`:

  ```
  All
    forwardDelegate selector sel
  {
    = nil;
  }
  ```

- (Skip this item if you don't understand it.) If the object implements the method

  ```
  InvocationResult
    forwardSelector selector sel
          arguments pointer pap;
  ```

  then that method is invoked, with `sel` being the selector of the message being forwarded, and `pap` a pointer to a va_list for the arguments in the message. The `InvocationResult` that is returned defines the values to be returned from the invocation that is being forwarded.

  Note that an object implementing a method `foo` is different from an object respoding `TRUE` when asked `respondsTo`. The latter can be overridden while the former is a direct check, which is much faster.

  This seemingly nasty low-level approach is used for fast dispatching of `too.RemoteProxy` method invocations and for invocations on curried `tom.Invocation` objects.

- With everything having failed so far, the whole invocation is packed into a newly created `Invocation` object, and sent to the receiver with a `forwardInvocation` method. The receiver can then decide what to do with it. The default implementation by the instance All raises a `program-condition` for the selector and target of the `Invocation`:

  ```
  InvocationResult
    forwardInvocation Invocation invocation
  {
  ```

```
  [[SelectorCondition
    for self class program-condition
    message "unrecognized selector"
    selector [invocation selector]]
   raise];
}
```

The `forwardInvocation` mechanism can also be used to mimic the functionality of the `forwardDelegate` method. If the `forwardDelegate` would return the object `my_delegate`, the following method would provide identical functionality:

```
InvocationResult
  forwardInvocation Invocation invocation
{
  = [invocation fireAt my_delegate];
}
```

Speed

Everything comes with a price, especially flexibility.

I've done some speed tests (on a PII/266, Debian 1.3.x, gcc 2.7.2.1) with various ways to invoke a method, with various number of arguments, with the results shown in the table below.

- *x* is a direct method invocation,

  ```
  [foo do (1, 2)];
  ```

- *p* is a perform,

  ```
  [foo perform selector ("(v)do(ii)") with (1, 2)];
  ```

- *d* is through a `forwardDelegate`. The invocation looks like that of *x*, but it is invoked on an object that does not implement the method, but does provide the following method (an instance of `Sub` does implement the desired method).

  ```
  Sub
    forwardDelegate selector s
  {
    = fwd_delegate;
  }
  ```

- *i* is through a `forwardInvocation`. Similar to *d*, but instead of `forwardDelegate`, the following method is implemented:

  ```
  InvocationResult
    forwardInvocation Invocation invocation
  {
    = [invocation fireAt fwd_delegate];
  }
  ```

Table 6-3. Speed of method invocation

how	#inv	time						
		0	1	2	3	4	5	6
x	10^8	12.58	12.93	12.58	12.53	13.69	14.00	13.91
p	10^7	11.15	11.77	12.74	13.80	14.88	15.63	17.32
d	10^7	11.62	12.75	13.75	15.11	15.81	16.61	18.52
i	10^6	15.96	17.44	17.80	18.14	18.51	18.87	19.22

* *In Table 6-3, using the `morerows` attribute causes jade/html to produce incorrect tables, and jade/tex to die.*

In Table 6-3, 'how' describes how the method is invoked, '#inv' is the number of invocations performed in the given time, and 'time' is the CPU time needed to run the test for the indicated number of arguments.

Note that for every `forwardInvocation` dispatch, an `Invocation` object and an `InvocationResult`is created, and approximately half the time taken by the test runs is spent in garbage collecting those objects. Also note that the mechanism underlying `perform with` and `forwardDelegate`, does not create `InvocationResult` objects for void methods, as was the case in the test.

The numbers come down to the following: in the time that you can do 1 `forwardInvocation`, you can do 10 `forwardDelegate` calls or invocations through `perform with`, and 100 direct method invocations.

I don't know if these numbers are good or bad; I don't have numbers to compare them with (maybe testing Objective-C Rhapsody on a PII/266?). It does show, that using `forwardDelegate` is much faster than having an `Invocation` object be created and forwarding that. Which is what it was supposed to do. (And it can probably be made much faster when using `__builtin_apply_args` in `trt_forward`, instead of going through `perform_args`.)

II. TOM: The Libraries

Chapter 7. The TOM Runtime Library

The TOM Runtime Library is the library that enables TOM programs to run. It contains the data structures needed for every method invocation; it performs object allocation, garbage collection, etc.

In C, all functionality of TOM and the TOM runtime can be obtained by including `<tom/util.h>`. This will also include the header generated by the resolver for the tom unit.

Program startup

From the start of a TOM program, the following sequence of actions takes place:

- If the program was not statically resolved, build the method dispatch tables and other runtime structures needed but not built by the resolver. When run, this installs what the GNU Objective-C runtime very nicely calls the 'premature' dispatch table for each object. Upon invocation of a method through said table, the actual dispatch table will be built and put in place.
- Collect the arguments to the program, excluding C's `argv[0]` into `tom.Runtime.all_arguments`, using `tom.ByteString` objects . Set the `tom.Runtime.program_name` and `tom.Runtime.long_program_name` from `argv[0]`.
- Initialize the `in`, `out`, and `err` streams of `tom.stdio`.
- Invoke the load imps, i.e. every class method with a signature matching `void load Array arguments`, with the arguments collected from the command line. A load imp is allowed to recognize options and remove them from the arguments. It is customary for such options to start with a colon (':') instead of a dash ('-').
- Store the array of remaining arguments in `tom.Runtime.arguments`.
- Invoke the following method of the `class tom.Runtime`:

  ```
  int
        start (All, selector) (object, sel)
    arguments Array arguments;
  ```

  where the `sel` is the selector for the main method to be invoked, which normally is `int main Array arguments`. The `object` is the receiver of this message, normally a class object.
- When the previous method invocation returns, all open streams are flushed and the value it returned it used as the exit code.

* *[Streams are not yet flushed.]*

C names for TOM types

Table 7-1 lists the C types to use for the TOM types.

Table 7-1. C names for TOM types

C	TOM
void	void
tom_byte	byte boolean
tom_char	char
tom_int	int
tom_long	long
tom_float	float
tom_double	double
void *	pointer
selector	selector
tom_object	*any object reference*

Selectors

In the runtime library, a selector is identifyable by a string which is called the selector's name. This name consists of the name parts and an abbreviated form of the argument and return types. The abbreviated type names are:

Table 7-2. Selector type encodings

char	type
v	void
o	boolean
b	byte
c	char
i	int
l	long
f	float
d	double
p	pointer
s	selector
r	object reference
x	dynamic

The following table shows examples: for several method declarations the name of the corresponding selector.

* *[Grammar desired.]*

Table 7-3. Example selector name encodings

method	selector
`int value;`	`"(i)value"`
`void setValue float d;`	`"(v)setValue(d)"`
`(Foo) bar (int, double) (a, b) with: int c = 0;`	`"(r)bar(id)with:(i)"`

* *[Actually, this is the future syntax. Currently, it is ambiguous, hence replaceable.]*

In C, a selector is defined by a struct; The `struct selector` is the direct implementation of the TOM selectors.

```
typedef struct selector
{
  unsigned int sel_id;

  struct name name;

  struct trt_selector_args *in, *out;
} *selector;
```

`sel_id`

This is the unique identity of a selector. All selectors have such an identity in the closed enumeration of all selectors. Every two selectors with identical name, argument types and return types have the same identity. If no dynamic loading has taken place, the following is true for any two selectors `a` and `b`:

* *[Actually, it isn't any longer.]*

```
a->sel_id == b->sel_id <-> a == b.
```

`name`

the name of the selector. This has two fields: `s` being the zero terminated string, and `len` being its length.

`in`
`out`

the selector argument descriptions for the arguments to (`in`) and return value from (`out`) methods denoted by this selector.

Message dispatching

This section explains how messages can be dispatched, i.e. how the implementation of a method can be invoked, given the message.

A method is translated by the compiler to a C function with essentially the same arguments as the method, with two mandatory additions and some additions depending on the return type of the method. For the method

```
(boolean, Any) next
```

defined for the `instance tom.Enumerator`, the C function implementing this is

```
tom_byte
i_tom_Enumerator_or_next (tom_object self,
                          selector cmd,
                          tom_object *ret1)
@end example
```

As can be seen, the first element of the tuple return type is the type returned by the implementation C function. Such a C function implementing a method always has two 'implicit' first arguments: `self` being the object receiving the message, and `cmd` being the message sent. Following these two are the 'normal' arguments to the method, which are none in this case. Finally, any of the remaining return values are to be returned in the pointer arguments supplied after the normal arguments. In the example, a pointer for returning the second tuple element is provided.

A message is dispatched by invoking the method implementation for the given `(self, cmd)` pair. The lookup of the implementation, i.e. the pointer to the C function, can be done in three different ways:

lookup

: Call the function `trt_lookup` with the receiver and the selector to be invoked (i.e. the values to be passed for `self` and `cmd`). `trt_lookup` returns a function pointer to the method implementation. Invoke that function with the arguments.

direct

: A direct lookup is equivalent to inlining the `trt_lookup` function. This is a rather unwise way of invoking a method as it considerably increases code size.

send

: When using `trt_send` to dispatch a message, the function `trt_send` is invoked with all the arguments to be passed to the method implementation. `trt_send` will perform the lookup and jump directly to the implementation.

These different dispatching mechanisms can be selected by an option to tomc. These options are not yet implemented in tesla, the new TOM compiler. When using tesla, they must be selected when configuring trt. They are described here for explanatory purposes; never implement any of these directly in your C code; the next section explains how to do that portably.

Sending is the preferred way of dispatching messages, though possibly not present on all TOM implementations as it involves an assembly language routine. Also note that there are dependencies of the applicability of some dispatching mechanisms. For example, it is impossible to use sending on dynamically loaded code on `hppa-hpux` machines. This is not a TOM feature but due to the interspace stubs needed by the hpux shared library interspace calls, added to the fact that a callee stores the return program counter in the stack frame of the caller.

When doing profiling on a TOM program, all code should really use the lookup way of dispatching instead of sending to dispatch. Otherwise, all methods will be reported to only invoke `trt_send`, and `trt_send` will be reported as the culprit which invoked every method and thus effectively void the use of the call graph.

Messaging from C

To invoke a method of a TOM object, use the `TRT_SEND` macro. For example, to retrieve the length of an array `a`:

```
tom_object a = ...;

tom_int len = TRT_SEND (, a, SEL (_i_length));
```

The second argument to `TRT_SEND` is the receiver of the message. The third argument is the selector to be sent. A selector is an invocation of the `SEL` macro, with as argument the selector's name with any nasty characters replaced by an underscore.

A more elborate example shows almost all pitfalls when using `TRT_SEND`. In this example the method `(int, float) split float f` is invoked, which in TOM would be written as

```
SomeClass receiver = ...;
float fractional, number = ...;
int integer;

(integer, fractional) = [receiver split number]
```

is invoked from C as follows:

```
tom_object receiver = ...;
tom_float fractional, number = ...;
tom_int integer;

integer = TRT_SEND ((tom_int (*) (tom_object, selector,
                                  tom_float, tom_float *)),
                    receiver, SEL (_if__split_f), number, &fractional);
```

The first argument to `TRT_SEND` is a cast to the type of the function actually being invoked. This cast is mandatory if the return type of the function (implementing the method) invoked is not a tom_int. The argument prototypes in the cast are mandatory when needed to prevent the C compiler from doing undesirable type conversions due to it not having seen a full prototype of the function being invoked. For instance, usually, tom_float is simply a float, which the compiler will promote to a double when passed as an argument for which the prototype has not been seen.

This example also shows that for a method returning a tuple, the first element of the tuple (or the first element thereof in case it is a tuple too (or...)) is actually returned from the C function implementation, and any remaining elements of the tuple returned are stored in variables the address' of which has been passed as 'invisible' trailing arguments.

More types

This section lists various types that are used by the runtime library.

struct name

```
struct name
{
  char *s;
  int len;
};
```

The `s` points to the zero-terminated C byte-string holding the name. The `len` is the length of `s`.

trt_selector_args

```
struct trt_selector_args
{
  int num;

  enum trt_type_encoding args[0];
};
```

A `trt_selector_args` describes the arguments to or return value from a method. In this context, all values are de-tupled and concatenated. Thus, a selector accepting an int and a float has the same argument description as a selector accepting a tuple (int, float). For the arguments, this excludes the implicit two first arguments, `self` and `cmd`.

`num`

The number of entries in `args`.

`args`

The description for each flat argument.

enum trt_type_encoding

```
enum trt_type_encoding
{
  TRT_TE_VOID,
  TRT_TE_BOOLEAN,
  TRT_TE_BYTE,
  TRT_TE_CHAR,
  TRT_TE_INT,
  TRT_TE_LONG,
  TRT_TE_FLOAT,
  TRT_TE_DOUBLE,
  TRT_TE_POINTER,
  TRT_TE_SELECTOR,
  TRT_TE_REFERENCE,
  TRT_TE_DYNAMIC,
};
```

The `trt_type_encoding` is used in the definition of argument and return types of selectors.

Functions

This section describes functions and macros defined by the TOM runtime, or any of its C header files.

byte_string_with_c_string

```
tom_object
byte_string_with_c_string (const char *s);
```

Return a newly allocated instance of `tom.ByteString` holding the characters from the zero-terminated string `s`. Obviously, the trailing zero is not contained in the returned `ByteString`.

byte_string_with_string

```
tom_object
byte_string_with_string (const char *s, int len);
```

Return a newly allocated instance of `tom.ByteString` holding the first `len` characters pointed to by `s`.

trt_assign_local_var

```
TRT_INLINE void *
trt_assign_local_var (void *object)
```

This function must be invoked if the `object` has just been assigned to a local variable and it is to live over a method invocation. Note that *this is only necessary iff* the stack protection policy implemented by the garbage collector (configured when building the TOM tools) is `SP_PROTECT` as opposed to `SP_MARK` (see `config/default.h` and `config/target.h`), so, probably (ahem), it isn't necessary.

* *[Obviously, the type of the argument should be tom_object instead of void *.]*

trt_assign_object_var

```
TRT_INLINE void *
trt_assign_object_var (void *object, void *value);
```

This function must be invoked if the object pointed to by `value` has just been assigned to an object variable of the `object`. This is needed in case the garbage collector performs non-atomic runs. Thus, if you're writing library code in C, you *must* use this function. The compiler outputs calls to this function if the flag `-fincremental-gc` is provided on the command line (@pxref{Invoking tomc}). This feature is not supported yet in tesla.

* *[Obviously, the type of the two arguments should be tom_object instead of void *. However, since debugging a TOM program currently means debugging the (not so unreadable, to the trained eye at least) C code output by the compiler, the compiler types each object in the output to its C struct (as far as the compiler can know the layout at compile time). Hence, void * is used in some places where really tom_object should be used, to avoid numerous casts or warnings.]*

trt_ext_address

```
void *
trt_ext_address (tom_object self, int extension_id);
```

`trt_ext_address` returns a pointer to the state information of the object `self` for the state introduced by the extension with the identity `extension_id`. *This is the only legitimate way to obtain a pointer to some state held by some object.* The only exception to this rule is the state information introduced by the State class (or instance), which, by definition, resides at offset 0 from `self`.

trt_selector_args_match

```
int
trt_selector_args_match (struct trt_selector_args *a,
                         struct trt_selector_args *b);
```

Return 1 if the number and types of the elements in `a` and `b` match, or 0 otherwise.

Normally, this test is very fast, since the resolver guarantees that for every pair of selector argument descriptions, `a` and `b`, if `trt_selector_args_match` returns `1`, the `a == b`. However, in the context of dynamic-loading, selector argument descriptions can guaranteed to be unique, thus making `trt_selector_args_match` slightly more expensive.

trt_selector_named

```
selector *
trt_selector_named (char *s, int len);
```

Return the selector whose name matches the name held in the first `len` bytes pointed to by `s`. Return `NULL` if such a selector does not exist.

trt_type_name

```
char *
trt_type_name (enum trt_type_encoding type);
```

Return a zero terminated C string holding the name of the `type`. If the `type` is not a valid value, the string `"<unknown type %d>"` is returned, with the numeric value of the `type` replacing the `%d`.

* *[In the case of an unknown type being returned this function leaks memory, since the string returned is malloced.]*

xmalloc

```
void *xmalloc (unsigned int n);
void *xcalloc (unsigned int n, unsigned int m);
void *xrealloc (void *p, unsigned int n);
void xfree (void *p);
```

Use these allocation manipulation routines instead of the x-less counterparts they are wrapping, since the ‘x’ is a clue on some machines.

Chapter 8. Unit `tom`

The `tom` unit is that standard TOM library.

File tom/All

class tom.All

The `All` class does not serve a purpose. It is the stateless `All` instance which is inherited by both the `State` class and instance, and, supposedly, by all objects not inheriting from `State`.

instance tom.All

inherits

State supers: `Conditions`, `Constants`

variables

```
const TRUE = !0;
```

The boolean truth.

```
const FALSE = !TRUE;
```

The boolean non-truth.

```
const YES = TRUE;
```

An alternative name for `TRUE`.

```
const NO = FALSE;
```

An alternative name for `FALSE`.

methods

```
String
  description;
```

Return a string informally describing this object.

This returns the result of having the receiver `write` itself into a new `String` which is subsequently returned.

```
boolean (result)
  eq All other
post
  self == other == result;
```

The selector equivalent of '==', i.e. the returned `result` is `TRUE` iff the receiving object and the `other` object are the same object.

The postcondition states that this method is not overridable.

```
boolean (result)
  equal id other
post
  self == other -> result;
```

Return `TRUE` when the receiving object considers itself equal to the other object. For instance, two `Number` objects holding the same value will return `TRUE`.

The receiving object should be able to assume the `other` object is of the same kind, or at least shares with it a common superclass, as in the case of, for instance, `CharString` and `ByteString` which are both subclasses of `String` and can compare with each other.

As stated by the postcondition, an object must be `equal` to itself. This knowledge may be used by a caller to prevent a method invocation.

```
int
  hash;
```

Return a hash value for the receiving object. The default implementation returns some bit pattern deduced from `self`.

Two distinct objects considering themselves `equal` should also return the same `hash` value.

```
int
  hashq;
```

Hash the address of the receiving object. For classes not redefining `hash`, this performs the same function.

```
id (self)
  self;
```

Return the receiving object.

```
deferred OutputStream
  write OutputStream s;
```

All objects, even classes, know how to (descriptively) write themselves to a stream.

A default implementation of this method is provided by the `State` class and instance.

```
deferred boolean
  classp;
```

Return `TRUE` iff the receiving object is a class object. An implementation for this method is provided by the `State` class and instance.

```
boolean
  isKindOf class (State) a_class;
```

Return `TRUE` iff the class of the receiving object is a subclass of the `class`.

```
deferred class (id)
  kind;
```

Return the class of the receiving object.

```
boolean
  respondsTo selector sel;
```

Return `YES` iff the selector `sel` can be safely sent to the receiver. The default implementation only checks whether the receiving object provides a direct implementation of the `sel`; any checking through an alternative `forwardDelegate` should be performed by the object itself.

```
All
  forwardDelegate selector sel;
```

Return an object of which the method indicated by the selector `sel` should be invoked. This method is invoked if the receiving object does not directly respond to `sel`. The default implementation returns `self`. The object returned could be a delegate which is to act upon behalf of the receiving object for the intended call of the selector `sel`.

```
InvocationResult
  forwardInvocation Invocation invocation;
```

Return the result of forwarding the `invocation`, for example by firing it at an appropriate object. The default implementation raises a `program-condition` `SelectorCondition`.

```
dynamic
  perform selector sel
        : Array arguments = nil;
```

Send the receiving object a message with the selector `sel` and the, possibly unboxed, `arguments`. The number of elements of `arguments` must match the number of arguments dictated by the selector.

Unboxing the arguments means that if an `int` argument is needed, the `int at int` method will be used to retrieve the argument from the `arguments`, possibly resulting in the object retrieved being asked for its `intValue`.

```
dynamic
  perform selector sel
     with dynamic arguments;
```

Send the receiving object a message with the selector `sel` and the `arguments`. The number of `arguments` must match the number of arguments dictated by the selector.

If the selector sel accepts more than one argument, arguments should be a tuple. The tuple-ization of the actual arguments to the selector sel and the elements of the arguments tuple is ignored.

```
Thread
  performInThread selector sel
             with dynamic arguments;
```

Like perform with but create a new thread for the performance. Return the newly created thread or nil upon failure.

```
boolean
  invocationp;
```

Return YES iff the receiving object is an Invocation. Only Invocation objects are supposed to return YES.

```
dynamic
  valueOfVariableNamed ByteString name;
```

Retrieve the value of the variable with the indicated name. If there is more than one variable with the same name and expected return type, the first is returned.

```
void
         setValue dynamic value
  ofVariableNamed ByteString name;
```

Set the value of the variable named name in the receiving object to the value. It is an error if the type of the value does not exactly match the actual type of the variable: no conversion is performed.

```
int
  typeOfVariableNamed String name
                 from Extension ext
pre
  [[self stateExtensions] memq ext] != nil;
```

Return the type of the variable named name as introduced by the extension ext. This returns one of the TYPEDESC_* Constants.

```
Any
  valueOfVariableNamed String name
                  from Extension ext
pre
  [[self stateExtensions] memq ext] != nil;
```

Return the boxed value of the variable named name as introduced by the extension ext.

```
Extension
  extensionNamed String name
      inherited: boolean check_supers = NO;
```

Return the `Extension` object of this object for the extension named `name`. If `name == nil`, the main extension is returned.

```
Indexed
  extensions;
```

Return an array of the extensions of the receiving object, not including the extensions introduced by superclasses.

```
Indexed
  allExtensions;
```

Return an array of all extensions of the receiving object. This includes the extensions introduced by superclasses.

```
Indexed
  stateExtensions;
```

Return an array of state introducing extensions of the receiving object. This includes the extensions introduced by superclasses.

```
void
  throw dynamic value;
```

Throw execution to the catch specified for the receiving object, returning the `value`. If the `value` is `void`, the default value for the type to be returned by the catch is returned.

```
void
  preconditionFailed selector sel;
```

This method is invoked for a failed precondition of a method invocation of the receiving object. The method is identified by the `selector sel`. The default implementation raises a `condition-condition SelectorCondition`.

Method precondition checking is enabled is the option `:cc-pre` is provided on the program's command line. The code for precondition checking is normally compiled in by the compiler. This code is omitted by passing the `-fno-checks` or `-fno-pre-checks` option to the compiler.

```
void
  postconditionFailed selector sel;
```

This method is invoked for a failed postcondition of a method invocation of the receiving object. The method is identified by the `selector sel`. The default implementation raises a `condition-condition SelectorCondition`.

Similar to precondition checking, postcondition checking is enabled by the `:cc-post` option on the command line of this program and not providing `-fno-checks` or `-fno-post-checks` to the compiler.

```
protected void
```

```
  unimplemented selector sel
       message: String message = nil;
```

Moan about the selector `sel` not yet having been implemented by the receiving object. This raises an `unimplemented SelectorCondition`.

```
protected void
  shouldNotImplement selector sel;
```

Contrary to what the inheritance tells you about the `selector sel` being invokable for the receiving object, that object thinks otherwise.

```
protected void
  subclassResponsibility selector sel;
```

Moan about the receiving object defining a method for the selector `sel`, but actually the implementation of the method by the object thinks it should be implemented by a subclass.

```
boolean
  consp;
```

Return `TRUE` iff the receiving object is a `Cons` cell. The default implementation returns `NO`.

```
OutputStream
  writeListElement OutputStream s;
```

Finish outputing the list, of which the receiving object is the tail, to the stream `s`. The default implementation writes itself as a dotted cdr at the end of the list.

```
deferred boolean
  persistent-coding-p;
```

Return `YES` iff the receiving object is a persistent object. This is significant for distributed objects, where class objects and `Selector` instances must be persistent across different invocations.

```
pointer
  address;
```

Return the address of the receiving object as a pointer. This is here solely to be able to print the address of objects, for debugging purposes.

```
boolean
  coding-permanent-object-p;
```

Return `YES` if the receiving object should be maintained in the permanent object store when coding. This does not matter for archiving; it makes a difference for DO. Class objects and `Selector`s return `TRUE` for this; the default implementation returns `FALSE`.

```
void
   dump (boolean, boolean) (allow_self, allow_simple)
```

```
  level int level;
```

Dump the graph of which the receiving object is the root to stderr.

```
void
  dump;
```

Like `void dump (boolean, boolean)`, allowing self/simple printing and doing infinite recursion.

```
boolean
  dump_simple_p;
```

Return `TRUE` iff the receiving object can be dumped simply. This will be true for class objects, strings, numbers, etc. This method is overridden by `dump_self_p`. The default implementation returns `FALSE`.

```
boolean
  dump_self_p;
```

Return `TRUE` iff the receiving object wants to dump itself instead of having its variables scrutinized. This is used by collection objects and others which employ `pointer` typed variables. The default implementation returns `FALSE`.

```
OutputStream
  dump_simple OutputStream s;
```

Dump the receiving object to the stream `s`, simply. This is only ever invoked if the object returns `YES` for `dump_simple_p`. The default implementation simply prints `self` to the stream.

```
protected void
  dumpSelf MutableKeyed done
    indent MutableByteString prefix
    simple boolean allow_simple
     level int level
        to OutputStream s;
```

Have the receiving object dump itself. Only ever invoked if it returns `TRUE` for `dump_self_p`. The default implementation invokes `shouldNotImplement`.

```
deferred protected void
    dump MutableKeyed done
  indent MutableByteString prefix
  simple boolean allow_simple
   level int level
      to OutputStream s;
```

Hard worker for `dump`.

```
boolean
  gc_dead_p;
```

Return `YES` iff the receiving object has not yet been marked alive during the current run of the garbage collector. Class objects are never dead.

```
void
  gc_mark;
```

Mark the receiving object as being alive. This method is only needed by the container garbage collection scheme.

This method is invoked during Garbage Collection. During GC, the `Runtime` library is running in panic mode. If anything goes wrong, for instance a condition is signaled or raised, the program will abort. Moral: be careful during garbage collection.

File tom/Array

class tom.Array

`Array` is the superclass of all arrays; it is an `Indexed` `Collection`

inherits

State supers: `State`, `Indexed`, `C`

instance tom.Array

variables

```
public int length;
```

The number of elements in the array.

```
pointer contents;
```

A pointer to the elements of this array.

methods

```
void
  dealloc;
```

Clean up the memory this array is using.

```
deferred int
  elementByteSize;
```

Return the size, in bytes, of the elements contained in this `Array`.

```
id
  initAsCopyOf id other;
```

Get the elements from the `other`, and invoke `[self initCopy]`.

```
id (self)
  initCopy;
```

Duplicate our `contents` since that is what we own.

```
id (self)
  initWith int n
        at pointer addr;
```

Initialize with the indicated pointer and integer for contents and length.

```
Any
  member All object;
```

Return the element contained in this `Array`, which is `equal` to the `object`.

```
Any
  memq All object;
```

Like `member`, but the element is identified on reference equality.

```
deferred (pointer, int) (address, number)
  pointerToElements (int, int) (start, len)
pre
  start >= 0 && len >= -1
post
  number >= 0 && !number == !address;
```

Return the `address` of the first element of the receiving array in the range `(start, len)`, and the `number` of elements in that range.

```
void
  makeVanishingElementsPerform Invocation inv;
```

Like `makeElementsPerform`, but allow the element currently messaged to vanish from this array.

File tom/Bag

class tom.Bag

A Bag is a `Keyed Collection`.

inherits

State supers: `HashTable`, `Keyed`

instance tom.Bag

methods

```
int
  at All object;
```

Return the number of times the element `key` is present in the bag.

```
Any
  at All object;
```

Return the `object` if present; `nil` otherwise.

```
Enumerator
  enumerator;
```

Undocumented.

```
id
  initWithEnumerator Enumerator e;
```

Undocumented.

class tom.MutableBag

inherits

State supers: `Bag`, `MutableKeyed`

instance tom.MutableBag

methods

```
void
    add All object
  count int num;
```

Add the `object` `num` times.

```
void
  add All object;
```

Add the `object`.

File tom/Block

class tom.Block

inherits

State supers: `State`, `Conditions`, `Enumerator`

variables

```
static boolean check_block_selectors;
```

methods

```
void
  load Array arguments;
```

Initialize the static control variables (only `check_block_selectors` up to now).

instance tom.Block

variables

```
pointer code;
```

Pointer to the actual code (a C function).

```
selector arguments;
```

The selector of the `eval` method of this block, which includes the formal argument and return types.

```
pointer variables;
```

If this block employs block variables, the `variables` points to a struct holding those variables.

```
pointer environment;
```

Pointer to the local variables of the enclosing method that are referenced from this block. This is not set when the block does not reference its environment; it is cleared when the environment is exited. A block that uses its environment checks upon entry to its eval method that the environment is still available.

```
pointer block_description;
```

A description of the variables in `variables`.

methods

```
id (self)
  initWithCode pointer block_c_function
       trigger selector full_arguments
       context pointer context
```

```
    variables (pointer, pointer) (vars, desc);
```

Designated initializer.

```
protected boolean (mismatch)
  arguments_fail (selector, selector) (formal, actual)
pre
  formal != actual;
```

Return `FALSE` if the arguments in the `formal` and `actual` selectors match, or match enough. Raise a `program-condition` for a mismatch (and return `TRUE`). The precondition dictates that the fast check should be done autonomously.

```
dynamic
  eval dynamic arguments;
```

Generic `eval` method. Faster versions, which are specialized on their arguments, are below.

```
void
  eval;
```

The first of many (similar) type-specific `eval` methods.

```
int (result)
  eval int a1;
```

Undocumented.

```
void
  eval int a1;
```

Undocumented.

```
Any (result)
  eval All a1;
```

Undocumented.

```
Any (result)
  eval;
```

Undocumented.

```
(boolean, Any) (remaining, value)
  next;
```

Undocumented.

```
void
  dealloc;
```

Release the memory used by this block.

```
void
  gc_mark_elements;
```

Mark the block variables if needed.

```
void
  invalidate;
```

Be informed that the block is going out of scope, invalidating the `environment`.

File tom/BucketDictElement

class tom.BucketDictElement

inherits

State supers: `BucketElement`

instance tom.BucketDictElement

variables

```
All key;
```

This bucket element's key.

```
public Any value;
```

The value in this bucket element.

methods

```
id
  initWith (All, All) (k, v);
```

Designated initializer.

```
void
  do Block block;
```

Apple the `block` to `value` and pass to `next`.

```
void
  doKeys Block block;
```

Apple the `block` to `key` and pass to `next`.

```
Any
  key;
```

Return the `key`, with a suitable type.

```
Any
  member All k;
```

Return the value associated with the key `k`, asking the `next` element if this element does not match.

The implementation by `BucketDictElement`, considers its `key` and returns its `value`.

```
Any
  member All k
   equal selector cmp;
```

Like member, but using the selector `cmp` to have the objects compare themselves.

```
Any
  memq All k;
```

Undocumented.

```
int
  add (All, All) (k, v);
```

Add the `(k, v)` pair to this bucket, if the key is not already present. Return the number by which this bucket's length has increased.

```
int
  addq (All, All) (k, v);
```

Add the `(k, v)` pair to this bucket, if the key is not already present. Return the number by which this bucket's length has increased.

```
(id, int)
  remove All k;
```

Remove the object with the key equal to `k`. Return the replacement for this element, and the number of bucket elements that were removed from this bucket list (max 1).

```
(id, int)
  removeq All k;
```

Remove the object with the identical key `k`. Return the replacement for this element, and the number of bucket elements that were removed from this bucket list (max 1).

```
void
  encodeUsingCoder Encoder coder;
```

Undocumented.

```
void
  initWithCoder Decoder coder;
```

Undocumented.

```
(id, int)
  gc_mark_values;
```

Starting with this bucket element, remove the objects of which the `value` is `gc_dead`. Return the replacement for this element, and the number of bucket elements that were removed from this bucket list.

```
(id, int)
  gc_mark_keys;
```

Similar to `gc_mark_values`, but consider the liveness of the `key` instead of the `value`.

```
int
  rehash;
```

Rehash the key of the receiving element.

```
int
  rehashq;
```

Rehashq the key of the receiving element.

File tom/BucketElement

class tom.BucketElement

inherits

State supers: `State`

instance tom.BucketElement

variables

```
public id next;
```

The next element in this bucket.

methods

```
void
  do Block block;
```

Apple the `block` to `self` and pass to `next`.

```
Any
  member All key
pre
  key != nil;
```

Return the value associated with the `key`, asking the `next` element if this element does not match.

The implementation by `BucketElement` considers itself to be both the key and the value.

```
Any
  member All key
   equal selector cmp
pre
  key != nil;
```

Like member, but using the selector `cmp` to have the objects compare themselves.

```
Any
  memq All key
pre
  key != nil;
```

Like `member`, but use reference equality instead of the `equal` method.

```
int
  addElement id elt;
```

Add the `elt` to this bucket, if it is not already present. Return the number by which this bucket's length has increased.

```
int
  addqElement id elt;
```

Add the `elt` to this bucket, if it is not already present. Return the number by which this bucket's length has increased.

```
(id, int) (replacement, decrease)
  remove id elt
post
  !decrease -> replacement == self;
```

Remove the `elt` from this bucket, if present. Return the number by which the length of this bucket has decreased, and the replacement remainder of the bucket list.

```
(id, int) (replacement, decrease)
  removeq id elt
post
  !decrease -> replacement == self;
```

Remove the elt from this bucket, if present. Return the number by which the length of this bucket has decreased, and the replacement remainder of the bucket list.

```
void
  resizing_feed HashTable ht;
```

For resizing a hashtable, feed this element and those following elements, to the hashtable using the hashtable's resizing_add.

```
void
  resizing_add id n;
```

While resizing a hashtable, accept a new next.

```
int
  rehash;
```

Rehash the key of the receiving element.

```
int
  rehashq;
```

Rehashq the key of the receiving element.

```
void
  encodeUsingCoder Encoder coder;
```

Undocumented.

```
void
  initWithCoder Decoder coder;
```

Undocumented.

```
(id, int)
  gc_mark_values;
```

Starting with this bucket element, remove those bucket elements of which the objects are gc_dead. Return the replacement for this element, and the number of bucket elements that were removed from this bucket list.

File tom/BucketIntDictElement

class tom.BucketIntDictElement

inherits

State supers: BucketElement

instance tom.BucketIntDictElement

variables

```
public int key;
```

This bucket element's key.

```
public Any value;
```

The value in this bucket element.

methods

```
id
  initWith (int, All) (k, v);
```

Designated initializer.

```
Any
  member int k;
```

Return the value associated with the key `k`, asking the `next` element if this element does not match.

The implementation by `BucketIntDictElement`, considers its `key` and returns its `value`.

```
Any
  memq int k;
```

Undocumented.

```
int
  add (int, All) (k, v);
```

Add the `(k, v)` pair to this bucket, if the key is not already present. Return the number by which this bucket's length has increased.

```
int
  addq (int, All) (k, v);
```

Add the `(k, v)` pair to this bucket, if the key is not already present. Return the number by which this bucket's length has increased.

```
int
  rehash;
```

Return the integer key.

```
int
  rehashq;
```

Return the integer key.

```
(id, int)
  remove int k;
```

Remove the object with the key `k`. Return the replacement for this element, and the number of bucket elements that were removed from this bucket list (max 1).

```
(id, int)
  gc_mark_values;
```

Starting with this bucket element, remove the objects of which the `value` is `gc_dead`. Return the replacement for this element, and the number of bucket elements that were removed from this bucket list.

File tom/BucketPDictElement

class tom.BucketPointerDictElement

inherits

State supers: `BucketElement`

instance tom.BucketPointerDictElement

variables

```
public pointer key;
```

This bucket element's key.

```
public Any value;
```

The value in this bucket element.

methods

```
id
  initWith (pointer, All) (k, v);
```

Designated initializer.

```
Any
  member pointer k;
```

Return the value associated with the key `k`, asking the `next` element if this element does not match.

The implementation by `BucketDictElement`, considers its `key` and returns its `value`.

```
int
  add (pointer, All) (k, v);
```

Add the `(k, v)` pair to this bucket, if the key is not already present. Return the number by which this bucket's length has increased.

```
(id, int)
  gc_mark_values;
```

Starting with this bucket element, remove the objects of which the `value` is `gc_dead`. Return the replacement for this element, and the number of bucket elements that were removed from this bucket list.

```
int (code)
  rehash;
```

Rehash the pointer key.

```
int (code)
  rehashq;
```

Rehash the pointer key.

```
(id, int)
  remove pointer k;
```

Remove the object with the key `k`. Return the replacement for this element, and the number of bucket elements that were removed from this bucket list (max 1).

File tom/BucketSetElement

class tom.BucketSetElement

inherits

State supers: `BucketElement`

instance tom.BucketSetElement

variables

```
Any value;
```

The key/value in this bucket element.

methods

```
void
  do Block block;
```

Apple the `block` to `value` and pass to `next`.

```
Any
  key;
```

Undocumented.

```
id
  initWith All v;
```

Designated initializer.

```
Any
  member All key;
```

Return the value associated with the `key`, asking the `next` element if this element does not match.

The implementation by `BucketSetElement`, considers its `value` as the both the key and the value.

```
Any
  member All key
   equal selector cmp;
```

Like member, but using the selector `cmp` to have the objects compare themselves.

```
Any
  memq All key;
```

Undocumented.

```
int
  add All key;
```

Add the `key` to this bucket, if it is not already present. Return the number by which this bucket's length has increased.

```
int
  addq All key;
```

Add the `key` to this bucket, if it is not already present. Return the number by which this bucket's length has increased.

```
(id, int) (replacement, decrease)
  remove All key;
```

Remove this bucket, if it holds the `key`. Return the number by which the length of this bucket list has decreased, and the replacement remainder of the bucket list.

```
(id, int) (replacement, decrease)
  removeq All key;
```

Remove this bucket, if it holds the `key`. Return the number by which the length of this bucket list has decreased, and the replacement remainder of the bucket list.

```
void
  encodeUsingCoder Encoder coder;
```

Undocumented.

```
void
  initWithCoder Decoder coder;
```

Undocumented.

```
void
  gc_mark_containers;
```

Tell the `value` to `gc_container_mark_elements`.

```
(id, int)
  gc_mark_values;
```

Starting with this bucket element, remove the objects of which the `value` is `gc_dead`. Return the replacement for this element, and the number of bucket elements that were removed from this bucket list.

```
int
  rehash;
```

Rehash the key of the receiving element.

```
int
  rehashq;
```

Rehashq the key of the receiving element.

File tom/Bundle

class tom.Bundle

inherits

State supers: `State`, `Constants`

variables

```
static Bundle main;
```

The main bundle, i.e. the bundle describing the program and the units.

```
static MutableIndexed units_path;
```

The path as registered by the units.

methods

```
OutputStream (s)
  help OutputStream s
  done MutableKeyed done;
```

Hook for responding to command line argument `:help`.

```
pointer
    loadUnit String unit
  fromObject String object;
```

Load the `unit` from the `object`, returning the underlying operating system handle upon success, or the NULL pointer upon failure. An `error` is signaled in case of the latter. The `unit` is the name of the unit supposedly contained in the `object`. This unit, when present, will be resolved.

```
pointer
  load String object;
```

Derive the unit name from the `object` name and invoke `loadUnit fromObject`.

```
void
  load MutableArray arguments;
```

Accept `:main-bundle-dir` option and allocate the `main` bundle if found.

```
String
    locate-file String file
      extension String ext
  with-version: String version = nil;
```

Forward this to the main bundle.

```
instance (id)
  main;
```

Return the main bundle, creating it iff necessary.

```
void
  registerUnitDirectory String dir;
```

Register the `dir` as to contain resources for one of the units.

instance tom.Bundle

variables

```
public String directory;
```

The directory.

```
pointer handle;
```

Iff not the null pointer, the handle (in the underlying operating system) to the code loaded for this bundle. Iff it is the null pointer, the code has not (yet) been loaded.

methods

```
id (self)
  init String d;
```

Undocumented.

```
String
    locate-file String file
      extension String ext
  with-version: String version = nil;
```

Locate the `file` for the `version` in this bundle. If not found, search the main bundle. Iff still not found, it is searched for in the registered unit directories.

The extension `ext`, if not `nil`, is appended to the `file`, with a dot (`.`) in between.

File tom/ByteArray

class tom.ByteArray

Like the `CharArray`, the `ByteArray` is a particular kind of `Array`, which is here for abstraction purposes, but which is never actually used, since the `ByteString` holds the same kind of state, but provides much more functionality.

inherits

State supers: `Array`

instance tom.ByteArray

methods

```
protected id
  initWithEnumerator Enumerator e;
```

Undocumented.

```
Any
  at int index;
```

Undocumented.

```
byte
```

```
  at int index;
```

Undocumented.

```
char
  at int index;
```

Undocumented.

```
int
  at int index;
```

Undocumented.

```
long
  at int index;
```

Undocumented.

```
float
  at int index;
```

Undocumented.

```
double
  at int index;
```

Undocumented.

```
int
  elementByteSize;
```

Undocumented.

```
int
  hash;
```

Undocumented.

```
(pointer, int)
  pointerToElements (int, int) (start, len);
```

Undocumented.

```
int
  writeRange (int, int) (start, len)
        into MutableByteArray destination
          at int position;
```

Undocumented.

```
void
  encodeUsingCoder Encoder coder;
```

Undocumented.

```
void
  initWithCoder Decoder coder;
```

Undocumented.

```
class (State)
  mutableCopyClass;
```

Return the `MutableByteArray` class.

File tom/ByteStream

class tom.ByteStream

Instances of the `ByteStream` class are an abstraction of the UNIX file descriptors.

inherits

State supers: `Descriptor`, `InputOutputStream`

instance tom.ByteStream

methods

```
byte
  read;
```

Read a byte and return it, raising an exception on end-of-file or error.

```
int
  read;
```

Read a byte and return it, returning `EOF` on end-of-file.

```
int
  readRange (int, int) (start, num)
       into MutableByteArray buffer;
```

Have the `buffer` read at most `num` bytes from the receiving `ByteStream` starting at `start`.

```
void
  write byte b;
```

Undocumented.

```
int
  write byte b;
```

Undocumented.

```
int
  writeBytes int num
        from pointer address;
```

Undocumented.

File tom/ByteString

class tom.ByteString

A `ByteString` is a `String` and a `ByteArray`, which can do all kinds of nice string-like things.

Requesting a `substring` of a `ByteString` results in a `ByteSubstring` to be returned. This will mimic a `ByteString` as much as possible, including hashing, equality, uniquing, printing, copying, etc, but they do not share a common superclass between `String` and `ByteString`.

In the future, the `ByteString` instance actual functionality could be put into a `ByteFullstring`, enabling the `ByteSubstring` to actually become a subclass of `ByteString`...

inherits

State supers: `ByteArray`, `String`, `C`, `Constants`

variables

```
static CharacterEncoding default_encoding;
```

The default character encoding for `ByteString` instances.

Never refer this variable directly; always ask the string (even if it is `self`) for its `encoding`. A normal `ByteString` will then return this `default_encoding`.

methods

```
OutputStream
  help OutputStream s
  done MutableKeyed done;
```

Output information on the `ByteString` unit arguments.

```
void
  load MutableArray arguments;
```

Set the default byte encoding. If it is not specified on the command line, `iso-8859-1` will be used.

Before this method is invoked by the runtime library, the `default_encoding` will be a `USASCIIEncoding`.

```
void
  switchToEncoding String name;
```

Switch to the encoding with the `name`, moaning if it fails (without changing the current encoding).

instance tom.ByteString

methods

```
char
  at int index;
```

Return the Unicode character for the byte at `index`.

```
(pointer, int)
  byteStringContents;
```

Undocumented.

```
boolean
  equal String other;
```

Undocumented.

```
int
  hashRange (int, int) (start, len);
```

Undocumented.

```
boolean
  equalByteString ByteString other;
```

Undocumented.

```
boolean
  equalCharString CharString other;
```

Undocumented.

```
boolean
  equalUniqueString UniqueString other;
```

Undocumented.

```
protected id (self)
  init (pointer, int) (p, num);
```

Initialize the newly allocated instance with the num bytes at p. The receiving instance will 'own' the memory at p.

```
id (self)
  initCopy (pointer, int) (p, num);
```

Initialize the newly allocated instance with a copy of the num bytes at p.

```
MutableByteString
  mutableSubstring (int, int) (start, len);
```

Return a new instance of the receiver's mutableCopyClass, initialized with a substring from the receiver's range (start, len).

```
String
  substring (int, int) (start, len);
```

Undocumented.

```
UniqueByteString
  uniqueString;
```

Undocumented.

```
OutputStream
  write OutputStream s;
```

Undocumented.

```
class (State)
  mutableCopyClass;
```

Return the MutableByteString class.

```
CharacterEncoding
  encoding;
```

Return the encoding of the receiving ByteString. The default implementation returns the default_encoding.

```
String
  stringByDecoding String encoding_name;
```

Undocumented.

```
String
  stringByDemapping CharArray demap;
```

Undocumented.

```
boolean
  isAlpha byte b;
```

Return `TRUE` the character denoted by the byte `b` in the encoding of the receiving string is a letter.

```
boolean
  isDigit byte b;
```

Return `TRUE` the character denoted by the byte `b` in the encoding of the receiving string is a digit.

```
boolean
  isLower byte b;
```

Return `TRUE` the character denoted by the byte `b` in the encoding of the receiving string is a lowercase letter.

```
boolean
  isPunct byte b;
```

Return `TRUE` the character denoted by the byte `b` in the encoding of the receiving string is a punctuation character.

```
boolean
  isSpace byte b;
```

Return `TRUE` the character denoted by the byte `b` in the encoding of the receiving string is a space character.

```
boolean
  isUpper byte b;
```

Return `TRUE` the character denoted by the byte `b` in the encoding of the receiving string is a uppercase letter.

```
byte
  toLower byte b;
```

Return the lower-case version of the byte `b`, according to the encoding of the receiving string. If the character is not in upper-case, it is returned unharmed.

```
byte
  toUpper byte b;
```

Return the upper-case equivalent of the byte `b`, according to the encoding of the receiving string. If the character is not in lower-case, it is returned unharmed.

```
int
  digitValue byte b;
```

Return the value equivalent of the byte `b`, for which this string should return `TRUE` when asked `isDigit`.

```
int
  alphaValue byte b;
```

Return the index of the letter `b` relative to the start of its letter range. Thus, 'a' returns 0, 'f' returns 5, etc.

```
id
  downcase;
```

This version of `downcase` overrides the implementation by `String`, since this one is faster due to avoiding the unnecessary conversion to/from Unicode.

```
id
  upcase;
```

Like `downcase`, this just is a faster implementation than the one provided by `String`.

File tom/ByteSubstring

class tom.ByteSubstring

A `ByteSubstring` is a substring of a constant `ByteString`. It tries to maskerade as one (even though it is certainly not an `Array`), possibly not perfect (yet).

inherits

State supers: `String`, `C`

methods

```
instance (id)
  with (int, int) (start, len)
    in ByteString string;
```

Undocumented.

instance tom.ByteSubstring

variables

```
ByteString string;
```

The string we're begin part of.

```
int start;
```

The start of us in our `string`.

```
public int length;
```

The length of us, which is never `< 0`.

methods

```
id
  init (int, int) (s, l)
    in ByteString str;
```

Designated initializer.

```
class (State)
  mutableCopyClass;
```

Return the `MutableByteString` class.

```
byte
  at int index;
```

Retrieve the `byte` at the `index`.

```
ByteNumber
  at int index;
```

Return the `ByteNumber` at the `index`.

```
Enumerator
  enumerator;
```

Return a restricted enumerator on the underlying string.

```
id
  initWithEnumerator Enumerator e;
```

Undocumented.

```
(pointer, int)
  pointerToElements (int, int) (begin, len);
```

Another low level access method.

```
ByteSubstring
  substring (int, int) (begin, len);
```

Return a new substring on our `string`---we do not cascade substrings.

```
MutableByteString
```

```
  mutableSubstring (int, int) (begin, len);
```

Undocumented.

```
(pointer, int)
  byteStringContents;
```

Low level access method.

```
boolean
  equal String other;
```

Undocumented.

```
int
  hash;
```

Undocumented.

```
int
  hashRange (int, int) (begin, len);
```

Undocumented.

```
boolean
  equalByteString ByteString other;
```

Undocumented.

```
boolean
  equalCharString CharString other;
```

Undocumented.

```
boolean
  equalUniqueString UniqueString other;
```

Undocumented.

```
UniqueByteString
  uniqueString;
```

Undocumented.

```
OutputStream
  write OutputStream s;
```

Undocumented.

File tom/C

class tom.C

The `C` class provides low-level memory manipulation functionality. With it, a lot of collection and string methods can be written in TOM instead of needing to be written in C.

inherits

Behaviour supers: `All`

methods

```
void
  free pointer address;
```

Release the memory at `address`.

```
pointer
  malloc int length;
```

Return a pointer to a newly allocated memory region of `length` bytes.

```
pointer
  calloc (int, int) (num, bytes);
```

Return a pointer to newly allocated and zeroed memory region of `num` elements of each `bytes` size.

```
pointer
  realloc (pointer, int) (address, length);
```

Return a pointer to the resized memory region at `address` which must hold `length` bytes. The address returned can differ from the previous `address`.

```
int
  memcmp (pointer, pointer, int) (one, other, length);
```

Return 0 iff the `length` bytes at `one` equal the bytes at `other`.

```
int
  memchr (pointer, int, int) (p, c, length);
```

Search the first `length` bytes from `s` for character `c`. Return the index into `s` at which `c` first occurs. If `c` is not present, return the value -1.

```
pointer
  memcpy (pointer, pointer, int) (to, from, length);
```

Copy the `length` bytes from `from` to `to`. Return `to`.

```
pointer
```

```
  memmove (pointer, pointer, int) (to, from, length);
```

Copy the `length` bytes from `from` to `to`, safely. Return `to`.

```
void
  bzero (pointer, int) (p, num);
```

Set the `num` bytes at `p` to 0.

instance tom.C

The `C` instance can be and is totally empty.

File tom/CharArray

class tom.CharArray

Like the `ByteArray`, the `CharArray` is a particular kind of `Array`, which is here for abstraction purposes, but which is never actually used, since the `CharString` holds the same kind of state, but provides much more functionality.

inherits

State supers: `Array`

instance tom.CharArray

methods

```
Any
  at int index;
```

Undocumented.

```
byte
  at int index;
```

Undocumented.

```
char
  at int index;
```

Return the char value at `index`. This is the elementary retrieval method for character arrays.

```
int
  at int index;
```

Undocumented.

```
long
  at int index;
```

Undocumented.

```
float
  at int index;
```

Undocumented.

```
double
  at int index;
```

Undocumented.

```
int
  elementByteSize;
```

Undocumented.

```
protected id
  initWithEnumerator Enumerator e;
```

Undocumented.

```
(pointer, int)
  pointerToElements (int, int) (start, len);
```

Undocumented.

```
class (State)
  mutableCopyClass;
```

Return the `MutableCharArray` class.

File tom/CharEncoding

class tom.CharacterEncoding

The `CharacterEncoding` class defines the interface of the byte and character encodings for predicates and conversions.

inherits

Behaviour supers: `All`

instance tom.CharacterEncoding

inherits

Behaviour supers: `All`

methods

```
deferred String
  name;
```

Return the name of this encoding.

```
deferred char
  decode byte b;
```

Return the decoded byte `b`, i.e. the Unicode character corresponding to the byte `b` in the receiving encoding.

```
deferred byte
  encode char c;
```

Return the byte encoding of the character `c`. If the byte equivalent of the character `c` does not exist in the receiving encoding, an `encoding-condition` is signaled, and the byte encoded is the `byte-Value` of the object returned, or 127 if `nil` is returned.

```
deferred boolean
  isAlpha byte b;
```

Return `TRUE` the character denoted by the byte `b` in the receiving encoding is a letter.

```
deferred boolean
  isDigit byte b;
```

Return `TRUE` the character denoted by the byte `b` in the receiving encoding is a digit.

```
deferred boolean
  isLower byte b;
```

Return `TRUE` the character denoted by the byte `b` in the receiving encoding is a lowercase letter.

```
deferred boolean
  isPunct byte b;
```

Return `TRUE` the character denoted by the byte `b` in the receiving encoding is a punctuation character.

```
deferred boolean
  isSpace byte b;
```

Return `TRUE` the character denoted by the byte `b` in the receiving encoding is a space character.

```
deferred boolean
  isUpper byte b;
```

Return `TRUE` the character denoted by the byte `b` in the receiving encoding is a uppercase letter.

```
deferred byte
  toLower byte b;
```

Return the lowercase version of the byte `b`, according to the receiving encoding. If the character is not in uppercase, it is returned unharmed.

```
deferred byte
  toUpper byte b;
```

Return the uppercase version of the byte `b`, according to the receiving encoding. If the character is not in lowercase, it is returned unharmed.

```
deferred int
  digitValue byte b;
```

Return the numeric value of the digit denoted by the byte `b` in the receiving encoding.

```
deferred int
  alphaValue byte b;
```

Return the index of the letter `b` relative to the start of its letter range. Thus, 'a' returns 0, 'f' returns 5, etc.

class tom.CharEncoding

An instance of the `CharEncoding` class maintains information on on a particular mapping for encoding a subset of Unicode characters to 8-bit bytes. An example of such mappings is `iso-8859-1`, which is the well known western european byte encoding, of which `USASCII` is a subset.

inherits

State supers: `State`, `Constants`, `Conditions`, `CharacterEncoding`

variables

```
static MutableDictionary encodings;
```

Currently known encodings.

methods

```
ByteArray
  loadBytes int num
       from String name
  extension String ext;
```

Load `num` bytes from the file with the `name` and the extension `ext` (sans dot). The full path of the file is obtained from the `main Bundle`.

```
instance (id)
  named String name;
```

Return the `CharEncoding` known as the `name`. This always succeeds, as a `CharEncoding` reads the resources it needs on demand.

instance tom.CharEncoding

variables

```
public String name;
```

The name of this encoding.

```
CharArray decoding;
```

The decoding map.

```
IntDictionary encoding;
```

The encoding map.

```
ByteArray to_lower;
```

The byte map for conversion to lower case within the encoding.

```
ByteArray to_upper;
```

The byte map for conversion to upper case within the encoding.

```
ByteArray to_title;
```

The byte map for conversion to title case within the encoding.

```
ByteArray is_digit;
```

The bitmap for testing whether a byte is a digit.

```
ByteArray is_letter;
```

The bitmap for testing whether a byte is a letter.

```
ByteArray is_lower;
```

The bitmap for testing whether a byte is lower case.

```
ByteArray is_punct;
```

The bitmap for testing whether a byte is a punctuation character.

```
ByteArray is_space;
```

Bitmap for space predicate.

```
ByteArray is_upper;
```

The bitmap for testing whether a byte is upper case.

methods

```
id
  init String n;
```

Designated initializer.

```
char
  decode byte b;
```

Return the decoded byte `b`, i.e. the Unicode character corresponding to the byte `b` in the receiving encoding.

```
CharArray
  decoding;
```

Return the `decoding` map, reading it iff necessary.

```
byte
  encode char c;
```

Return the byte encoding of the character `c`. If the byte equivalent of the character `c` does not exist in the receiving encoding, an `encoding-condition` is signaled, and the byte encoded is the `byteValue` of the object returned, or 127 if `nil` is returned.

```
IntDictionary
  encoding;
```

Return the `encoding` map, creating it from the `decoding` map if necessary.

```
protected ByteArray
  loadConversion String conversion;
```

Load and return the conversion table for the `conversion` of the receiving encoding.

```
protected ByteArray
  loadPredicateSet String predicate;
```

Load and return the predicate set for the `predicate` of the receiving encoding.

```
boolean
  isAlpha byte b;
```

Return `TRUE` the character denoted by the byte `b` in the receiving encoding is a letter.

```
boolean
  isDigit byte b;
```

Return `TRUE` the character denoted by the byte `b` in the receiving encoding is a digit.

```
boolean
  isLower byte b;
```

Return `TRUE` the character denoted by the byte `b` in the receiving encoding is a lowercase letter.

```
boolean
  isPunct byte b;
```

Return `TRUE` the character denoted by the byte `b` in the receiving encoding is a punctuation character.

```
boolean
  isSpace byte b;
```

Return `TRUE` the character denoted by the byte `b` in the receiving encoding is a space character.

```
boolean
  isUpper byte b;
```

Return `TRUE` the character denoted by the byte `b` in the receiving encoding is a uppercase letter.

```
byte
  toLower byte b;
```

Return the lowercase version of the byte `b`, according to the receiving encoding. If the character is not in uppercase, it is returned unharmed.

```
byte
  toUpper byte b;
```

Return the uppercase version of the byte `b`, according to the receiving encoding. If the character is not in lowercase, it is returned unharmed.

```
int
  digitValue byte b;
```

Return the numeric value of the digit denoted by the byte `b` in the receiving encoding.

```
int
  alphaValue byte b;
```

Return the index of the letter `b` relative to the start of its letter range. Thus, 'a' returns 0, 'f' returns 5, etc.

class tom.USASCIIEncoding

A replacement for a real `CharEncoding` used during program initialization.

inherits

State supers: `State`, `CharacterEncoding`

variables

```
static USASCIIEncoding shared;
```

The one and only `USASCIIEncoding` object.

methods

```
instance (id)
  shared;
```

Undocumented.

instance tom.USASCIIEncoding

methods

```
String
  name;
```

We're really a dummy, so we do not have a name. In fact, that is how we're recognized.

```
char
  decode byte b;
```

This is acceptable for iso-8859-1.

```
byte
  encode char c;
```

This is acceptable for iso-8859-1.

```
boolean
  isAlpha byte b;
```

Undocumented.

```
boolean
  isDigit byte b;
```

Undocumented.

```
boolean
  isLower byte b;
```

Undocumented.

```
boolean
  isPunct byte b;
```

Undocumented.

```
boolean
  isSpace byte b;
```

Undocumented.

```
boolean
  isUpper byte b;
```

Undocumented.

```
byte
  toLower byte b;
```

Undocumented.

```
byte
  toUpper byte b;
```

Undocumented.

```
int
  digitValue byte b;
```

Undocumented.

```
int
  alphaValue byte b;
```

Undocumented.

File tom/CharString

class tom.CharString

inherits

State supers: `CharArray`

instance tom.CharString

methods

```
boolean
  equal String other;
```

Undocumented.

```
boolean
  equalByteString ByteString other;
```

Undocumented.

```
boolean
  equalCharString id other;
```

Undocumented.

```
boolean
  equalUniqueString UniqueString other;
```

Undocumented.

```
protected id
  init (pointer, int) (p, num);
```

Initialize the newly allocated instance with the `num` characters at `p`. The receiving instance will 'own' the memory at `p`.

```
id (self)
  initCopy (pointer, int) (p, num);
```

Initialize the newly allocated instance with a copy of the `num` chars at `p`.

```
MutableCharString
  mutableSubstring (int, int) (start, len);
```

Undocumented.

```
CharString
  substring (int, int) (start, len);
```

Undocumented.

```
UniqueCharString
  uniqueString;
```

Undocumented.

```
class (State)
  classForCoder Encoder coder;
```

Undocumented.

```
class (State)
  mutableCopyClass;
```

Return the `MutableCharString` class.

File tom/Condition

class tom.Condition

inherits

State supers: `State`

methods

```
instance (id)
      for All object
    class ConditionClass condition_class
  message String msg;
```

Return a new `Condition` for the indicated circumstances.

instance tom.Condition

variables

```
public ConditionClass condition_class;
```

The condition class of the condition indicated by this `Condition`.

```
public Any object;
```

The object by/for which this condition was raised.

```
public String message;
```

The message explaing what actually happened.

```
public boolean raised;
```

Iff `TRUE`, this condition was raised, otherwise it was signaled.

methods

```
protected id
  initFor All o
    class ConditionClass cc
  message String msg;
```

Undocumented.

```
void
  raise;
```

Raise this condition; guaranteed never to return.

```
Any
```

```
  signal;
```

Signal this condition. If a handler performs a non-local break, this method does not return. If no handler is installed, `nil` is returned. If a handler returns something different from this condition, signaling is terminated and that value is returned.

```
OutputStream (s)
  writeFields OutputStream s;
```

Undocumented.

class tom.SelectorCondition

inherits

State supers: `Condition`

methods

```
instance (id)
       for All object
     class ConditionClass condition_class
   message String msg
  selector selector sel;
```

Return a new `SelectorCondition` for the indicated circumstances.

instance tom.SelectorCondition

variables

```
selector sel;
```

The selector which was sent to the `object`.

methods

```
protected id
   initFor All o
     class ConditionClass cc
   message String msg
  selector selector s;
```

Undocumented.

```
selector
  selector;
```

Return the selector, `sel`.

```
OutputStream (s)
```

```
  writeFields OutputStream s;
```

Undocumented.

File tom/ConditionClass

class tom.ConditionClass

Instances of the `ConditionClass` define the hierarchy of conditions as carried by `Condition` instances. Conditions classes could be real tom classes, but the features provided by said mechanism are too baroque for this purpose---only the inheritance is needed.

The tom condition class hierarchy does not employ multiple inheritance.

TOM conditions are evidently modelled after CL.

inherits

State supers: `State`

methods

```
instance (id)
  with instance (id) super_condition
  name ByteString name;
```

Undocumented.

instance tom.ConditionClass

variables

```
id super_condition;
```

Our super condition class. The super condition class of the top `condition` is `nil`.

```
public ByteString name;
```

Our descriptive name.

methods

```
protected id
  init id sc
  name ByteString nm;
```

Undocumented.

```
boolean
  isConditionSuper id other;
```

Return YES iff `other` is a super condition class of the receiving condition class.

File tom/Conditions

class tom.Conditions

The `Conditions` class is an instance-less non-static-state-less class providing predefined conditions. Not-predefined, i.e. user defined, conditions should be made available to the world through an extension of `Conditions`.

variables

`static All unhandled_condition_handler;`

The object informed of unhandled condition raises. Iff `nil`, the program aborts when a condition being raised is not handled. This feature is not yet implemented within the runtime.

`static selector unhandled_condition_selector;`

The selector of the message to be sent to the `unhandled_raise_handler`. This method accepts a single argument, which will be a `Condition`. This feature is not yet implemented within the runtime.

`static ConditionClass condition;`

Various (and numerous) condition classes, indented according to condition inheritance.

`static ConditionClass warning;`

`static ConditionClass unimplemented;`

`static ConditionClass encoding-condition;`

`static ConditionClass serious-condition;`

`static ConditionClass runtime-condition;`

`static ConditionClass runtime-fatal;`

```
static ConditionClass nil-receiver;

static ConditionClass unrecognized-selector;

static ConditionClass uncaught-throw;

static ConditionClass program-condition;

static ConditionClass unknown-class-condition;

static ConditionClass coding-condition;

static ConditionClass type-condition;

static ConditionClass lock-condition;

static ConditionClass condition-condition;

static ConditionClass error;

static ConditionClass file-error;

static ConditionClass stream-error;

static ConditionClass stream-eos;

static ConditionClass signal-condition;
```

```
static ConditionClass signal-hup;
```

```
static ConditionClass signal-int;
```

```
static ConditionClass signal-bus;
```

```
static ConditionClass signal-segv;
```

```
static ConditionClass float-condition;
```

```
static ConditionClass overflow-condition;
```

```
static ConditionClass underflow-condition;
```

instance tom.Conditions

File tom/Cons

class tom.Cons

inherits

State supers: `State`

methods

```
instance (id)
  with (All, All) (a, d);
```

Return a newly allocated instance with the `a` and `d` as the `car` and `cdr`, respectively.

```
instance (id)
  cons All a
     : All d = nil;
```

Return a newly allocated instance with the `a` and `d` as the `car` and `cdr`, respectively. The cdr `d` defaults to `nil`.

instance tom.Cons

variables

```
public Any car;
```

The element contained in this `Cons` cell, and the remainder of the list.

```
public Any cdr;
```

methods

```
protected id (self)
  init (All, All) (a, d);
```

Designated initializer.

```
boolean
  consp;
```

Return `YES`.

```
(Any, Any)
  decons;
```

Return the `(car, cdr)` in a tuple.

```
void
  set_car All c;
```

Set the `car` to the object `c`.

```
void
  set_cdr All c;
```

Set the `cdr` to the object `c`.

```
void
  encodeUsingCoder Encoder coder;
```

Encode the receiving object to the `coder`.

```
boolean
  equal id other;
```

Return `TRUE` if the receiving list is equal to the `other` list. Elements are compared with `equal`.

```
int (value)
  hash;
```

Use the `car` and `cdr` to compute a hash value for this `Cons` cell.

```
void
  initWithCoder Decoder coder;
```

Decode the receiving object from the `coder`.

```
id
  member All object;
```

Return the Cons cell whose car is `equal` to the `object`.

```
id
  memq All object;
```

Like `member`, but the element is identified on reference equality.

```
OutputStream
  write OutputStream s;
```

Output the list, of which the receiving `Cons` cell is the start, to the stream `s`.

```
OutputStream
  writeListElement OutputStream s;
```

Continue outputing the list, of which the receiving `Cons` cell is an element and not the head, to the stream `s`.

File tom/Constants

class tom.Constants

The constants used throughout the tom unit.

variables

`Trie` constants

```
const TRIE_PLAIN = 0;
```

The plain option indicates absence of other options.

```
const TRIE_REVERSED = 1;
```

Reverse the string before insertion or lookup.

`const TRIE_FOLD_CASE = 2;`

Ignore the case during a lookup; use lower-case characters during an insert.

`const TRIE_LOOKUP_PREFIX = 4;`

Do not require a full match in a lookup; the longest prefix will match with this option specified.

Open flags

`const FILE_EXIST_NOTHING = 1;`

If the file exists, do nothing (and return nil).

`const FILE_EXIST_RAISE = 2;`

If the file exists, raise a condition.

`const FILE_EXIST_TRUNCATE = 4;`

If the file exists and `output_p`, truncate it to zero length.

`const FILE_EXIST_SUPERSEDE = 8;`

If the file exists and `output_p`, a new version of the file will be created (by unlinking the old file first).

`const FILE_NOT_EXIST_NOTHING = 16;`

If the file does not exist, do nothing (and return nil). If only `input_p`, `nil` will also be returned if it can't be opened anyway.

`const FILE_NOT_EXIST_RAISE = 32;`

If the file does not exist, raise a condition.

`const FILE_NOT_EXIST_CREATE = 64;`

If the file does not exist and `output_p`, create it.

`const FILE_APPEND = 128;`

Every write will append to the end of the file.

`const FILE_MASK = 255;`

A mask for the above flags.

`const FILE_TYPE_NONEXISTENT = 0;`

Retrieving information about a file.

`const FILE_TYPE_OTHER = 1;`

`const FILE_TYPE_SOCKET = 2;`

`const FILE_TYPE_LINK = 3;`

`const FILE_TYPE_REGULAR = 4;`

`const FILE_TYPE_BLOCK = 5;`

`const FILE_TYPE_DIRECTORY = 6;`

`const FILE_TYPE_CHARACTER = 7;`

`const FILE_TYPE_FIFO = 8;`

Positioning a `SeekableStream`

`const STREAM_SEEK_SET = 0;`

Position absolute.

`const STREAM_SEEK_CUR = 1;`

Position relative to the current position.

`const STREAM_SEEK_END = 2;`

Position relative to the end of the file.

`TypeDescription` types.

`const TYPEDESC_VOID = 0;`

This value indicates the `void` type. The other values follow the same naming convention.

`const TYPEDESC_BOOLEAN = 1;`

`const TYPEDESC_BYTE = 2;`

```
const TYPEDESC_CHAR = 3;
```

```
const TYPEDESC_INT = 4;
```

```
const TYPEDESC_LONG = 5;
```

```
const TYPEDESC_FLOAT = 6;
```

```
const TYPEDESC_DOUBLE = 7;
```

```
const TYPEDESC_POINTER = 8;
```

```
const TYPEDESC_SELECTOR = 9;
```

```
const TYPEDESC_REFERENCE = 10;
```

```
const TYPEDESC_DYNAMIC = 11;
```

```
const TYPEDESC_NUM = 12;
```

This is not a real type; it merely denotes the number of `TYPEDESC_` values.

instance tom.Constants

File tom/DCons

class tom.DCons

inherits

State supers: `Cons`

methods

```
instance (id)
  with (All, All, All) (a, d, b);
```

Undocumented.

instance tom.DCons

variables

```
public Any cbr;
```

The back pointer.

methods

```
void
  set_cbr All c;
```

Undocumented.

```
boolean
  dconsp;
```

Undocumented.

```
protected id (self)
  init (All, All, All) (a, d, b);
```

Undocumented.

```
(All, All, All)
  dedcons;
```

Undocumented.

```
void
  unlink;
```

Undocumented.

class tom.ListEnumerator

inherits

State supers: `State`, `Enumerator`

methods

```
instance (id)
  with DList l;
```

Undocumented.

instance tom.ListEnumerator

variables

```
Cons cell;
```

The current cell.

methods

```
id (self)
  init Cons c;
```

Designated initializer.

```
(boolean, All)
  next;
```

Undocumented.

class tom.DList

inherits

State supers: `MutableOrdered`

methods

```
instance (id)
  new;
```

Undocumented.

instance tom.DList

variables

```
public DCons head;
```

```
public DCons tail;
```

methods

```
Enumerator
  enumerator;
```

Undocumented.

```
int
  length;
```

Undocumented.

```
void
  add All object;
```

Undocumented.

```
void
  empty;
```

Undocumented.

```
DCons
  cellAtIndex int index;
```

Undocumented.

```
void
  set All object
   at int index;
```

Undocumented.

```
void
  swap All object
    at (int, int) (i, j);
```

Undocumented.

```
void
  reverse (int, int) (start, len);
```

Undocumented.

```
void
  reverse;
```

Fast and easy method for the simplest case.

```
void
  pushHeadCons DCons cell;
```

Push the cell to the front of the list.

```
void
  pushTailCons DCons cell;
```

Push the cell to the back of the list.

```
void
  pushHead All object;
```

Push the element to the front of the list.

```
void
  pushTail All object;
```

Push the element to the back of the list.

```
void
  removeCons DCons cell;
```

Remove a cell from the list. The cell should be a member.

```
All
  popHead
pre
  head != nil;
```

Pop the element from the head of the list.

```
All
  popTail
pre
  head != nil;
```

Pop the element from the tail of the list.

```
All
  first
pre
  head != nil;
```

First element of the list.

```
All
  last
pre
  head != nil;
```

Last element of the list.

```
boolean
  dlistp;
```

Return TRUE.

File tom/Date

class tom.Date

The Date class implements absolute times using doubles to represent the number of seconds passed since a certain reference date. Internally the time at which the Date class was initialized is used as reference. As absolute reference the first instant of January 1, 2001 is used. All gregorian calculation functions use an absolute date which is the number of days since the Gregorian date December 31, 1 BC.

inherits

State supers: `State`

variables

```
const EPSILON = 1e-06;
```

For two dates to be considered equal they should be no further apart than `EPSILON`.

```
const OFFSET_DISTANT_FUTURE = 1e+100;
```

```
const OFFSET_DISTANT_PAST = -OFFSET_DISTANT_FUTURE;
```

```
const SECONDS_PER_DAY = 86400.0;
```

```
const ABSOLUTE_REFDATE = 730486;
```

The number of days from the imaginary Gregorian date Sunday, 31 december 1 BC to our reference date (January 1 2001).

```
static public double relative_offset;
```

Some offset from the reference date relative to which all Date instances maintain their notion of time. This is set in `load`, ensuring a high accuracy of dates near the moment in time during which this program is running.

```
static public Date distant_future;
```

A date in the very far future and a date in the very far past.

```
static public Date distant_past;
```

methods

```
double (now)
  relativeTimeIntervalSinceNow
post
  now > 0.0;
```

Return the number of seconds after `relative_offset` it is now.

```
double
  timeIntervalSinceReferenceDate;
```

Return the number of seconds after the absolute reference date it is now. This number is negative for dates before the first instant of January 1, 2001.

```
(int, double)
  absoluteAndSecondsOfTimeInterval double ti;
```

Return the absolute date and the seconds passed in that day for a time interval since the reference date.

```
int
  absoluteFromGregorian (int, int, int) (year, month, day);
```

The number of days elapsed between the Gregorian date 12/31/1 BC and `(year, month, day)`. The Gregorian date Sunday, December 31, 1 BC is imaginary.

```
int
  absoluteFromIso (int, int, int) (year, week, day);
```

The number of days elapsed between the Gregorian date 1 BC December 31 and DATE. The 'ISO year' corresponds approximately to the Gregorian year, but weeks start on Monday and end on Sunday. The first week of the ISO year is the first such week in which at least 4 days are in a year. The ISO commercial DATE has the form `(year, week, day)` in which `week` is in the range 1..52 and `day` is in the range 0..6 (1 == Monday, 2 == Tuesday, ..., 0 == Sunday). The Gregorian date Sunday, December 31, 1 BC is imaginary.

```
int
  dayNameOnOrBefore (int, int) (day_name, absolute);
```

Returns the absolute date of the `day_name` on or before `absolute`. `day_name`==0 means Sunday, `day_name`==1 means Monday, and so on.

Note: Applying this function to `absolute`+6 gives us the `day_name` on or after an absolute day d. Similarly, applying it to `absolute`+3 gives the `day_name` nearest to `absolute`, applying it to `absolute`-1 gives the `day_name` previous to `absolute`, and applying it to `absolute`+7 gives the `day_name` following `absolute`.

```
int
  dayNumber (int, int, int) (year, month, day);
```

Return the day number within the year of the date `(year, month, day)`. For example, dayNumber (1, 1, 1987) returns the value 1, while dayNumber (12, 31, 1980) returns 366.

```
int
  dayOfWeekOfAbsolute int absolute;
```

Return the Gregorian day of the week for `absolute` where 0==Sunday, 1==Monday, ..., 6==Saturday.

```
(int, int, int)
  gregorianFromAbsolute int date;
```

Compute the list (month, day, year) corresponding to the absolute DATE. The absolute date is the number of days elapsed since the (imaginary) Gregorian date Sunday, December 31, 1 BC.

```
boolean
  isLeapYear int year;
```

Return `TRUE` iff year is a Gregorian leap year.

```
(int, int, int)
  isoFromAbsolute int absolute;
```

Compute the 'ISO commercial date' corresponding to the `absolute`. The ISO year corresponds approximately to the Gregorian year, but weeks start on Monday and end on Sunday. The first week of the ISO year is the first such week in which at least 4 days are in a year. The ISO commercial date has the form (year week day) in which week is in the range 1..52 and day is in the range 0..6 (1 = Monday, 2 = Tuesday, ..., 0 = Sunday). The absolute date is the number of days elapsed since the (imaginary) Gregorian date Sunday, December 31, 1 BC.

```
int
  lastDayOfMonth int month
            year int year;
```

Return the last day of the month `month` of the year `year`.

```
void
  load MutableArray arguments;
```

Perform class initialization.

```
Date
  now;
```

Return a date instance representing this moment.

```
protected double
  relativeTimeIntervalOfAbsoluteAndSeconds (int, double) (absolute, seconds);
```

Return the absolute date and the seconds passed in that day for a time interval since the reference date.

```
double
  timeIntervalOfAbsoluteAndSeconds (int, double) (absolute, seconds);
```

Return the absolute date and the seconds passed in that day for a time interval since the reference date.

instance tom.Date

variables

```
double relative_ti;
```

methods

```
int
  compare id other;
```

Returns -1 if the receiver is earlier than `other` 0 if the difference is smaller that `EPSILON` and 1 if the receiver is after `other`.

```
id
  dateWithOffset double ti;
```

Return a new instance initialized at `ti` seconds after the receiver.

```
Date
  earlierDate Date other;
```

Return `other` if it is earlier than the receiver, return the receiver otherwise.

```
boolean
  equals id d;
```

Return `TRUE` iff the receiver is within `EPSILON` seconds of `d`.

```
protected id
  init double d;
```

Designated initializer.

```
id
  init;
```

Initialize with the current time.

```
id
  initWithTimeIntervalSinceNow double ti;
```

Initialize with `ti` seconds after the current time.

```
id
  initWithTimeIntervalSinceReferenceDate double ti;
```

Initialize with `ti` seconds after the absolute reference date January 1, 2001.

```
Date
  laterDate Date other;
```

Return `other` iff it is later than the receiver, return the receiver otherwise.

```
protected double
  relativeTimeInterval;
```

Return the number of seconds after `relative_offset`

```
double
  timeIntervalSinceDate Date d;
```

Return the number of seconds passed since `d`. This number is negative if the receiver is earlier than `d`.

```
double
  timeIntervalSinceNow;
```

Return the number of seconds passed since now. This number is negative for dates before now.

```
double
  timeIntervalSinceReferenceDate;
```

Return the number of seconds passed since the absolute reference date. This number is negative for all dates before the first instant of January 1 2001.

```
OutputStream
  write OutputStream s;
```

Print this date in a human readable format, relative to GMT.

```
void
  encodeUsingCoder Encoder coder;
```

Undocumented.

```
void
  initWithCoder Decoder coder;
```

Undocumented.

File tom/Descriptor

class tom.Descriptor

A `Descriptor` is the abstraction of the UNIX file descriptor.

inherits

State supers: `State`, `Conditions`

instance tom.Descriptor

variables

```
public int descriptor;
```

The file descriptor. This will be -1 if we're not actively open.

methods

```
void
  close;
```

Close this descriptor. If it succeeds, every read, write, or other operation afterwards will certainly fail. If it fails, this may not be the case.

```
void
  dealloc;
```

Close the `descriptor` if it is not `-1`.

```
id
  init;
```

Invoke `[self init -1]` to avoid ever closing file descriptor 0 by accident.

```
protected id
  init int fd;
```

Designated initializer: Initialize with `fd` as the `descriptor`.

```
int
  type-of-file;
```

Return one of the `FILE_TYPE_*` constants for the receiving File. Signal a `file_error` and return `FILE_TYPE_NONEXISTENT` if not open.

File tom/Dictionary

class tom.DictionaryContainer

A `DictionaryContainer` is a class which can be inherited by `Dictionary`-like objects, to allow them to be a container, with respect to their value objects. It is a seperate class, for inheritance by the `PointerDictionary` class.

inherits

State supers: `Container`

instance tom.DictionaryContainer

class tom.ObjectDictionary

An `ObjectDictionary` is a `Dictionary` mapping objects to objects. It is the superclass of `Dictionary` and `EqDictionary`. The latter hash the key objects on their address, and uses pointer equivalence.

inherits

State supers: `HashTable`, `Mapped`, `DictionaryContainer`

instance tom.ObjectDictionary

methods

```
void
  doKeys Block block;
```

Evaluate the `block` for each key.

```
Enumerator
  enumerator;
```

Return a `valueEnumerator`.

```
Any
  member All object;
```

Invoke `HashTable`'s implementation.

```
Any
  memq All object;
```

Invoke `HashTable`'s implementation.

```
Enumerator
  keyEnumerator;
```

Return an enumerator on the keys of this dictionary.

```
DictionaryEnumerator
  valueEnumerator;
```

Return an enumerator on the values of this dictionary.

```
OutputStream
  write OutputStream s;
```

Undocumented.

class tom.Dictionary

inherits

State supers: `ObjectDictionary`

instance tom.Dictionary

class tom.MutableDictionary

inherits

State supers: `Dictionary`, `MutableHashTable`, `MutableMapped`

instance tom.MutableDictionary

methods

```
void
  remove All key;
```

Remove the mapping for the `key`.

```
void
  add All object;
```

Undocumented.

```
void
  set All value
   at All key
pre
  value != nil && key != nil;
```

Undocumented.

class tom.DictionaryEnumerator

inherits

State supers: `HashTableEnumerator`, `MapEnumerator`

instance tom.DictionaryEnumerator

variables

```
redeclare BucketDictElement elt;
```

methods

```
(boolean, Any, Any) (valid, k, v)
  next;
```

Undocumented.

class tom.DictionaryValueEnumerator

inherits

State supers: `DictionaryEnumerator`

instance tom.DictionaryValueEnumerator

methods

```
(boolean, Any) (valid, object)
  next;
```

Undocumented.

File tom/DoubleArray

class tom.DoubleArray

inherits

State supers: `Array`

methods

```
instance (id)
  with dynamic elements;
```

Take the `double` arguments and craft a new array.

instance tom.DoubleArray

methods

```
protected id
  initWithEnumerator Enumerator e;
```

Undocumented.

```
Any
  at int index;
```

Undocumented.

```
byte
  at int index;
```

Undocumented.

```
char
  at int index;
```

Undocumented.

```
double
  at int index;
```

Undocumented.

```
long
  at int index;
```

Undocumented.

```
int
  at int index;
```

Undocumented.

```
float
  at int index;
```

Undocumented.

```
int
  elementByteSize;
```

Undocumented.

```
(pointer, int)
  pointerToElements (int, int) (start, len);
```

Undocumented.

```
class (State)
  mutableCopyClass;
```

Return the `MutableDoubleArray` class.

File tom/EqDictionary

class tom.EqDictionary

inherits

State supers: `ObjectDictionary`, `EqHashTable`

instance tom.EqDictionary

methods

```
id
  initWithEnumerator Enumerator e;
```

Undocumented.

class tom.MutableEqDictionary

inherits

State supers: `EqDictionary`, `MutableEqHashTable`, `MutableMapped`

instance tom.MutableEqDictionary

methods

```
void
  remove All key;
```

Remove the mapping for the `key`.

```
void
  set All value
   at All key
pre
  value != nil && key != nil;
```

Undocumented.

class tom.WeakKeyMutableEqDictionary

The WeakKeyMutableEqDictionary is identical to a MutableEqDictionary, except that when it is a container, the references to the keys are weak, whereas in the case of an ordinary MutableEqDictionary the value references are weak.

inherits

State supers: `MutableEqDictionary`

instance tom.WeakKeyMutableEqDictionary

methods

```
void
  gc_container_mark_elements;
```

Almost identical to `MutableHashTable`'s implementation, but the bucket elements are asked to `gc_mark_keys` instead of `gc_mark_values`.

File tom/EqHashTable

class tom.EqHashTable

inherits

State supers: `HashTable`

instance tom.EqHashTable

methods

```
Any
  at All key;
```

Return the `key` if present, or `nil` otherwise.

```
void
  resizing_add BucketElement elt;
```

Undocumented.

class tom.MutableEqHashTable

inherits

State supers: `EqHashTable`, `MutableHashTable`

instance tom.MutableEqHashTable

methods

```
void
  add BucketElement elt;
```

Undocumented.

File tom/EqSet

class tom.EqSet

inherits

State supers: `Set`, `EqHashTable`

instance tom.EqSet

class tom.MutableEqSet

inherits

State supers: `EqSet`, `MutableEqHashTable`, `MutableSet`, `Container`

Retrieve a (any) object from the set. This removes an element from the set and returns it.

instance tom.MutableEqSet

methods

```
void
  add All object;
```

Add the `object` to the receiving set.

```
void
  remove All object;
```

Remove `elt` from the receiving set, if present.

```
void
  gc_mark_containers
pre
  [self isContainer];
```

This method is invoked by the garbage collector to have the set containing all containers make those containers mark their elements. This method relies on the container containing the containers be itself a container.

File tom/Extension

class tom.Extension

The `Extension` class represents the runtime structures for class extensions. All classes have at least one extension, the main extension, which defines the behaviors and the state for that class. All other extensions are defined by the programmer as extensions to the class.

inherits

State supers: `State`, `Conditions`

variables

```
static MutableArray extensions;
```

All extensions, here to be protected against gc.

methods

```
instance (id)
  new pointer p;
```

Designated allocator. Do not use `alloc` or plain `new`.

instance tom.Extension

variables

```
pointer rti;
```

The runtime structure describing this extension.

```
Indexed method_selectors;
```

The selectors for this extension's methods.

```
Indexed var_names;
```

The names of this extension's variables.

methods

```
void
  dealloc;
```

An Extension should never be deallocated. This method raises, which is a panic during garbage collection...

```
boolean
  implements selector sel;
```

Return YES if this extension provides an implementation for the selector sel.

```
protected id (self)
  init pointer r;
```

Designated initializer.

```
class (State)
  meta;
```

Return the class object to which this extension belongs.

```
String
  name;
```

Return the name of this extension. This will be nil for the main extension of a class.

```
Indexed
  methods;
```

Return the selectors for the methods in this extension.

```
boolean
  hasState;
```

Return whether or not this extension defines variable additions.

```
Indexed
  variables;
```

Return the names of the variables in this extension.

```
int
  typeOfVariableNamed String name
                   in All object;
```

Return the type of the variable named name. The return value will correspond with one of the TYPE-DESC_* constants defined on the Constants class.

```
Any
  valueOfVariableNamed String name
                    in All object;
```

Undocumented.

```
void
         setValue dynamic value
  ofVariableNamed ByteString name
               in All object;
```

Undocumented.

```
OutputStream
  writeFields OutputStream s;
```

Undocumented.

```
dynamic
  perform selector sel
       on All object
     with dynamic arguments;
```

The equivalent of `perform with` where the method invoked is defined by this extension instead of the receiving object. Obviously, the `object` should actually have this extension as one of its extensions, i.e. `[object isKindOf [self meta]]` should be a precondition (and a postcondition too, but we're not interested after the fact).

```
dynamic
  perform selector sel
       on All object
        : Array arguments = nil;
```

Undocumented.

File tom/File

class tom.File

The File class offers an abstraction from files in the filesystem.

inherits

State supers: `ByteStream`, `SeekableStream`, `Constants`, `Conditions`

variables

```
const OPEN_INPUT = 256;
```

```
const OPEN_OUTPUT = 512;
```

methods

```
String
            basename String filename
  without-extension: String ext = nil;
```

Return the `filename` removing any directory component, and removing the extension `ext` if it matches and is not `nil`.

```
String
  current_directory;
```

Return the current directory, as a directory name (tailing slash).

```
void
  set_current_directory String directory;
```

Set the current `directory`. This raises a `file-error` when problems arise.

```
String
  directory-of-file String name;
```

Return the directory-name of the directory containing the file named `filename`.

```
String
  expand-filename String filename
     relative-to: String directory = nil;
```

Expand the `filename` relative to the `directory`. If `directory` is `nil`, expansion is relative to the current working directory.

```
String
  express-filename String filename
       relative-to String directory;
```

Express the `filename` in terms relative to the `directory`.

```
String
  filename-as-directory String filename;
```

Return the `filename` as the name of a directory.

```
MutableArray
  filenames-in-directory String dir_name;
```

Return the filenames in the directory named `dir_name`.

```
String
  locate-file String file
   along-path Indexed path;
```

Return the filename of the `file` somewhere along the `path`. Return `nil` if it could not be found.

```
boolean
  file-exists String name;
```

Return `YES` iff the file with the `name` exists.

```
int
  type-of-file String name
  follow-link: boolean follow_link = YES;
```

Return one of the `FILE_TYPE_*` constants.

```
instance (id)
     open String name
   input: boolean input_p = FALSE
  output: boolean output_p = FALSE
   flags: int action = 0;
```

Return a new `File`.

```
instance (id)
        open String name
   alongPath Indexed path
     subdir: String subdir1 = nil
  subsubdir: String subdir2 = nil;
```

Search for the file along the `path`.

For the subdirectories `subdir1` and `subdir2`, when not nil, the following attempts are made for a `dir` in the `path`: `dir`, `dir/subdir2`, `dir/subdir1`, and `dir/subdir1/subdir2`.

```
void
  remove String name;
```

Remove the file or directory with the `name`.

instance tom.File

variables

```
public String name;
```

The name of the file.

```
int flags;
```

If it is to be reopenend, these are the flags indicating how to do so.

methods

```
protected id
   init String n
  flags int f;
```

Designated initializer.

```
String
  directoryName;
```

Return the name of the directory containing our file.

```
id
  reopen;
```

Reopen the file according to the `flags`.

```
int
  type-of-file;
```

Return one of the `FILE_TYPE_*` constants for the receiving File. If the file is not open, the file is tested as if the file were open, i.e. following links.

```
OutputStream
  writeFields OutputStream s;
```

Undocumented.

```
long
  length;
```

Return the length of the file.

```
long
  position;
```

Return the current file position.

```
void
       seek long offset
  relative: int whence = STREAM_SEEK_SET;
```

Position the file ‘pointer’.

File tom/FloatArray

class tom.FloatArray

inherits

State supers: `Array`

methods

```
instance (id)
  with dynamic elements;
```

Take the `float` arguments and craft a new array.

instance tom.FloatArray

methods

```
protected id
  initWithEnumerator Enumerator e;
```

Undocumented.

```
Any
  at int index;
```

Undocumented.

```
byte
  at int index;
```

Undocumented.

```
char
  at int index;
```

Undocumented.

```
float
  at int index;
```

Undocumented.

```
long
  at int index;
```

Undocumented.

```
int
  at int index;
```

Undocumented.

```
double
  at int index;
```

Undocumented.

```
int
  elementByteSize;
```

Undocumented.

```
(pointer, int)
  pointerToElements (int, int) (start, len);
```

Undocumented.

```
boolean
  elementsLowerThan FloatArray fa;
```

Returns `TRUE` if every element in `self` is lower than the corresponding element in `fa`.

```
FloatArray
  minor FloatArray fa;
```

Undocumented.

```
class (State)
  mutableCopyClass;
```

Return the `MutableFloatArray` class.

File tom/HashTable

class tom.HashTable

inherits

State supers: `State`, `Keyed`

variables

```
const GOLDEN_BITS = -1640531527;
```

Thirty-two bits by which to multiply a hashvalue to make all bits, more or less, significant.

This (unsigned) value is (1 << 32) * frac (0.5 + 0.5 * sqrt (5)). The value of which the fractional part is taken is the golden ratio. (0x9e3779b9).

instance tom.HashTable

variables

```
public int length;
```

The number of stored objects.

```
int size_shift;
```

The 2log of the number of buckets.

```
MutableObjectArray buckets;
```

The buckets.

methods

```
id (self)
  init;
```

Designated initializer.

```
void
  empty;
```

Remove all elements from the table.

```
void
  do Block block;
```

Evaluate the `block` for each object element in this `HashTable`, by passing this method to the `BucketElements`.

```
Any
  at All key;
```

Return the `key` if present, or `nil` otherwise.

```
Any
  member All object;
```

A different name for `at`.

```
Any
  member All key
   equal selector cmp;
```

Like `member`, but the elements are compared using the selector `cmp`.

```
Any
  memq All key;
```

Like `member`, but the element is identified on reference equality.

```
void
```

```
  encodeUsingCoder Encoder coder;
```

Undocumented.

```
void
  initWithCoder Decoder coder;
```

Undocumented.

```
protected void
  adjust_length int inc
pre
  inc != 0;
```

Adjust the length of the hashtable, resizing if necessary.

```
protected void
  resize int new_shift;
```

Undocumented.

```
void
  resizing_add BucketElement elt;
```

Undocumented.

class tom.HashTableContainer

The HashTableContainer class is just a HashTable which knows how to mark its elements as a container. It is intended to be inherited by various class employing the HashTable class as a superclass for implementation reuse.

inherits

State supers: `HashTable`, `Container`

instance tom.HashTableContainer

methods

```
void
  gc_container_mark_elements;
```

Undocumented.

class tom.MutableHashTable

inherits

State supers: `HashTableContainer`, `MutableKeyed`

instance tom.MutableHashTable

methods

```
void
  add BucketElement elt;
```

Add `elt` to the receiving hashtable.

```
void
  remove BucketElement elt;
```

Remove `elt` from the receiving hashtable, if present.

class tom.HashTableEnumerator

inherits

State supers: `State`, `Enumerator`

methods

```
instance (id)
  with Indexed b;
```

Undocumented.

instance tom.HashTableEnumerator

variables

```
Indexed buckets;
```

The array the dictionary uses to store the buckets.

```
int next;
```

The next bucket index we shall use.

```
BucketElement elt;
```

The bucket element we're looking at.

methods

```
protected id
  init Indexed b;
```

Undocumented.

```
protected boolean
  next;
```

Update `elt` to point to the next bucket element.

File tom/Heap

class tom.Heap

inherits

State supers: `State`, `MutableCollection`

instance tom.Heap

variables

```
MutableArray elements;
```

The array used to store the elements.

```
public int length;
```

The number of elements. This is equal to `[elements length]`.

```
public mutable selector compare_selector;
```

The selector used to have two elements compare themselves. If this isn't set, `i_compare_r` will be used.

```
boolean max_heap;
```

If this is `TRUE`, this is a heap of which the root node is the largest. Otherwise, the root is the smallest node.

methods

```
id
  init;
```

Invoke `[self init TRUE]`.

```
id (self)
  init boolean root_is_max;
```

Designated initializer.

```
boolean
  dump_simple_p;
```

Return `NO`.

```
Any
  extract_min
pre
  !max_heap;
```

Extract the root of the heap, which must be a heap storing the minimum value at its root.

```
int
  index_of_element All element;
```

Return the index of the `element` in this heap. Used by elements that do not remember their own index.

```
Any (object)
  min
pre
  !max_heap;
```

Return the minimum value of the heap, which must be a heap storing the minimum value at its root. Return `nil` if the heap is empty.

```
Any
  extract_max
pre
  max_heap;
```

Extract the root of the heap, which must be a heap which stores the maximum value at its root.

```
Any (object)
  max
pre
  max_heap;
```

Return the maximum value of the heap, which must be a heap storing the maximum value at its root. Return `nil` if the heap is empty.

```
Any (object)
  extract_root
pre
  length > 0
post
  length == old length - 1;
```

Extract and return the root `object` of the heap.

```
Any (object)
  root
pre
  length > 0;
```

Return the root `object` of the heap, without extracting it.

```
void
  remove Comparable elt;
```

Remove the `elt` from the receiving heap. The `elt` should be an element of the heap.

```
void
  add Comparable object
post
  length == 1 + old length;
```

Add the `object` to this Heap.

```
void
  addElementsFromEnumerator Enumerator e;
```

Undocumented.

```
void
  empty;
```

Remove all elements.

```
Enumerator
  enumerator;
```

Return an enumerator on the elements of this heap. Note that the order of the elements returned is undefined.

```
protected void
  build_heap;
```

Build a heap from the elements already in `elements`.

```
int
  compare (Comparable, Comparable) (one, other);
```

Let `one` compare itself with the `other`, either using the `compare_selector`, or, if not set, the `int compare id` method.

```
protected void
  heapify int index;
```

Heapify from the node at index `i` (which is off by 1 compared to the index of the element in the `elements` array).

File tom/HeapElement

class tom.HeapElement

inherits

State supers: `State`, `Comparable`

instance tom.HeapElement

variables

```
int heap_index;
```

The index of the element within the `Heap` it is stored. This index is 1 more than the index in the `elements` `Array` of the `Heap`. If this element is not part of a heap, this index is 0.

methods

```
void
  set_index int index
    in_heap Heap heap;
```

Set the `heap_index` to `index`, without checking the `heap`.

```
int
  index_in_heap Heap heap;
```

Return the `heap_index`, assuming a correct `heap`.

File tom/IntArray

class tom.IntArray

inherits

State supers: `Array`

methods

```
instance (id)
  with dynamic elements;
```

Undocumented.

instance tom.IntArray

methods

```
protected id
  initWithEnumerator Enumerator e;
```

Undocumented.

```
Any
  at int index;
```

Undocumented.

```
byte
  at int index;
```

Undocumented.

```
char
  at int index;
```

Undocumented.

```
int
  at int index;
```

Undocumented.

```
long
  at int index;
```

Undocumented.

```
float
  at int index;
```

Undocumented.

```
double
  at int index;
```

Undocumented.

```
int
  elementByteSize;
```

Undocumented.

```
int
  hash;
```

Undocumented.

```
(pointer, int)
```

```
  pointerToElements (int, int) (start, len);
```

Undocumented.

```
class (State)
  mutableCopyClass;
```

Return the MutableIntArray class.

File tom/IntDictionary

class tom.IntDictionary

A IntDictionary maps an int to an object reference.

inherits

State supers: HashTable

instance tom.IntDictionary

methods

```
Any
  at int key;
```

Undocumented.

```
Enumerator
  enumerator;
```

Return a valueEnumerator.

```
Enumerator
  valueEnumerator;
```

Return an Enumerator on the values stored in this dict.

class tom.MutableIntDictionary

inherits

State supers: IntDictionary, HashTableContainer

instance tom.MutableIntDictionary

methods

```
void
  freeze;
```

Undocumented.

```
void
  remove int key;
```

Remove the mapping for the `key`.

```
void
  set All value
   at int key
pre
  value != nil;
```

Associate the `value` with the `key`.

File tom/IntegerRangeSet

class tom.IntegerRangeSet

The `IntegerRangeSet` is good at holding ranges of integers. Ranges are stored in an unbalanced tree. The number of ranges dictates the memory usage by the set. Testing for membership is O(log n) where n is the number of ranges.

inherits

State supers: `State`, `Enumerable`

instance tom.IntegerRangeSet

variables

```
IntegerRangeSetNode root;
```

The root of the tree holding information.

methods

```
boolean
  add int v;
```

Add `v` to the receiving set. Return `FALSE` if it was already present; `TRUE` otherwise.

```
boolean
  equal id other;
```

Undocumented.

```
boolean
  equalIntegerRangeSetNode IntegerRangeSetNode other;
```

Undocumented.

```
id
  intersectionWith id other;
```

Return a new set being the intersection of the receiving set and the `other` set.

```
boolean
  isEmpty;
```

Undocumented.

```
boolean
  isSubsetOf id other;
```

Return `YES` if the receiving set is a subset of the `other` set.

```
(boolean, int) (non_empty, value)
  highestPresent;
```

Return the highest value present in the set.

```
(boolean, int) (non_empty, value)
  lowestPresent;
```

Return the lowest value present in the set.

```
boolean
  member int v;
```

Return `TRUE` iff the set contains `v`.

```
int
  nextNonPresent int i;
```

Return the smallest element >= `i` that is not yet in the set.

```
(boolean, int)
  nextPresent int i;
```

Return the smallest element > `i` that is contained in the tree, preceded by whether such an element actually is in the tree.

```
int
  previousNonPresent int i;
```

Return the largest element <= `i` that is not yet in the set.

```
(boolean, int)
  previousPresent int i;
```

Return the largest element <= `i` that is in the set.

```
boolean (b)
  remove int v;
```

Remove `v` from the set, returning `TRUE` if it was actually contained.

```
void
  shiftFrom int i;
```

Increase the value of all elements in the tree >= `i` by one.

```
(boolean, int)
  smallestElement;
```

Return the smallest value contained, preceded by whether we're not empty.

```
void
  uniteWith id other;
```

Modify the receiving set by adding the elements from the other set.

```
OutputStream
  write OutputStream s;
```

Undocumented.

```
Enumerator
  enumerator;
```

Undocumented.

```
protected id
  initWithEnumerator Enumerator e;
```

Undocumented.

class tom.IntegerRangeSetNode

inherits

State supers: `State`

instance tom.IntegerRangeSetNode

variables

```
id left;
```

The left and right subtrees.

```
id right;
```

```
int offset;
```

The offset from our parent.

```
int size;
```

The number of integers in this node.

methods

```
boolean
  add int v;
```

Add `v` to the tree rooted at the receiving node. Return `FALSE` if it was already present; `TRUE` otherwise.

```
boolean
  equalIntegerRangeSetNode id other;
```

Undocumented.

```
id
  init (int, int) (o, s);
```

Designated initializer.

```
int
  highestPresent;
```

Return the highest value present in the set.

```
int
  lowestPresent;
```

Return the lowest value present in the set.

```
boolean
  member int v;
```

Return `TRUE` iff the set contains `v`.

```
int
  nextNonPresent int i;
```

Return the smallest element >= `i` that is not yet in the tree.

```
(boolean, int)
  nextPresent int i;
```

Return the smallest element > `i` that is contained in the tree, preceded by whether such an element actually is in the tree.

```
int
  previousNonPresent int i;
```

Return the largest element <= `i` that is not yet in the tree.

```
(boolean, int)
  previousPresent int i;
```

Return the largest element in the tree which is smaller than `i`.

```
(id, boolean)
  remove int v;
```

Remove `v` from the receiving tree, returning the modified tree, and `TRUE` if it was actually removed.

```
void
  shiftFrom int i;
```

Increase the value of all elements in the tree >= `i` by one.

```
int
  smallestElement;
```

Return the smallest value contained.

```
OutputStream
   write OutputStream s
  offset int i;
```

Undocumented.

```
protected (id, id, int, int)
  dissect;
```

Return the guts of this object.

```
protected (id, int)
  mergeLRL int v;
```

Merge the subtree rooted at this node, to accomodate the value `v`, returning the modified tree and the extra size for our parent node. Our parent actually holds the value `v` (as the first value).

```
protected (id, int)
```

```
  mergeRLL int v;
```

Merge the subtree rooted at this node, to accomodate the value `v`, returning the modified tree and the extra size for our parent node. Our parent actually holds the value `v` (as the last value).

```
protected void
  offset int n;
```

Adjust the `offset` by `n`.

```
protected void
  set_right id n;
```

Set the `right` node.

```
protected void
  setRightMost id r;
```

Set the right most node of the receiving tree to `r`.

class tom.IntegerRangeSetNodeEnumerator

inherits

State supers: `State`, `Enumerator`

instance tom.IntegerRangeSetNodeEnumerator

variables

```
IntegerRangeSetNode root;
```

The root of the tree of nodes.

```
int previous;
```

The previous integer value retrieved.

methods

```
id
  init IntegerRangeSetNode r;
```

Undocumented.

```
(boolean, Number)
  next;
```

Undocumented.

```
(boolean, int) (valid, value)
```

```
  next;
```

Undocumented.

File tom/Invocation

class tom.Invocation

An Invocation is an object holding a target object, a selector, and arguments to the selector. Thus, an Invocation holds everything needed to send a message. An Invocation can be fired at its target (with the fire method), or fired after retargeting (using the fireAt method).

An Invocation is incomplete when not all arguments needed to send the message have been specified. An incomplete Invocation can be fired in two different ways. First, by invoking fireWith and supplying values for the remaining arguments. Second, by invoking on the Invocation a method completing it. For example, if an Invocation x of the method void with int a do int b only has a value for the argument a, then invoking [x do 23] will (temporarily) complete the Invocation and send the full message (with do) to its target.

inherits

State supers: State

methods

```
instance (id)
   for selector sel
   to: All target = nil
  with dynamic arguments;
```

Create a, potentially incomplete, invocation.

```
instance (id)
  for selector sel
  to: All target = nil
    : Indexed arguments = nil;
```

Create a, potentially incomplete, invocation.

```
instance (id)
    of selector sel
   to: All target = nil
  with dynamic arguments;
```

Create a complete invocation. This raises a program-condition if the resulting invocation is incomplete.

```
instance (id)
```

```
    of selector sel
  to: All target = nil
    : Indexed arguments = nil;
```

Create a complete invocation. This raises a `program-condition` if the resulting invocation is incomplete.

```
instance (id)
     of selector sel
    to: All target = nil
  using pointer ap;
```

Create an invocation. It shall be complete. This method is primarily intended to be used by `Proxy` in its forwarding from `forwardSelector arguments`.

Arguments are to be retrieved from the `va_list` pointed to by `ap`, i.e. `va_arg (*ap, ...)`.

instance tom.Invocation

variables

```
InvocationResult result;
```

The result of the most recent invocation, or `nil` if we haven't fired yet, or have fired with a void return type (of the fire method).

```
pointer invocation;
```

The underlying invocation structure.

methods

```
protected id
  init pointer i;
```

Designated initializer.

```
boolean
  isComplete;
```

Return `TRUE` iff the receiving invocation is complete, i.e. is has all the arguments needed and can be fired directly with `fire` or `fireAt`.

```
selector
  selector;
```

Return this invocation's selector.

```
Any
  target;
```

Return this invocation's target.

```
protected InvocationResult
  forwardSelector selector sel
        arguments pointer ap;
```

Forward the selector `sel` with the arguments pointed to by the `va_list` pointed to by `ap`. Return the result of the invocation.

Only the incoming arguments from `*ap` will be retrieved, so that subsequent `va_arg` invocations on `*ap` can retrieve the outgoing argument pointers.

This method is invoked by the runtime library in an attempt to forward a message not directly implemented by the receiver. This method is used since it is faster than a `forwardInvocation`.

```
boolean
  invocationp;
```

Return `YES`.

```
InvocationResult
  fire;
```

Perform the invocation. If invoked repeatedly, the invocation will be performed repeatedly.

```
void
  fire;
```

Similarly, but avoid the creation of an `InvocationResult`. The `result` of the receiving invocation is set to `nil`.

```
InvocationResult
  fireAt All target;
```

Perform the invocation after setting the receiver of this invocation to `target`.

```
void
  fireAt All target;
```

Similarly, but avoid the creation of an `InvocationResult`. The `result` of the receiving invocation is set to `nil`.

```
InvocationResult
  fireWith dynamic arguments;
```

Perform the invocation resulting from completing this invocation with the `arguments`. Return the result. The receiving invocation will remain incomplete.

```
void
  fireWith dynamic arguments;
```

Similarly, but avoid the creation of an `InvocationResult`. The `result` of the receiving invocation is set to `nil`.

```
Any
  objectAfterFire;
```

Shortcut to fire and return the first element of the result as an object.

```
Any
  objectOfResult;
```

Shortcut to retrieve the first element of the result as an object. This fires if needed.

```
InvocationResult
  result;
```

If the invocation has been fired at least once, return the (most recent) result. Otherwise, fire and return the result.

```
TypeDescription
  resultTypeDescription;
```

Return the type description of the result from this invocation.

```
void
  encodeUsingCoder Encoder coder;
```

Undocumented.

```
void
  initWithCoder Decoder coder;
```

Undocumented.

```
protected void
  encodeToCoder Encoder coder;
```

Private method to do the hard work of encoding all information carried by this invocation.

```
protected void
  decodeFromCoder Decoder coder;
```

Private method to do the hard work of decoding all information from the `coder` into the invocation.

```
void
  dealloc;
```

Deallocate the structure underlying the invocation.

```
void
  gc_mark_elements;
```

Mark the objects in this invocation's arguments.

File tom/InvocationResult

class tom.InvocationResult

inherits

State supers: `State`, `Constants`

methods

```
protected instance (id)
  with pointer result;
```

Return a freshly allocated instance with the `result`, which is a `struct trt_invocation_result`.

instance tom.InvocationResult

variables

```
pointer values;
```

The actual result.

methods

```
protected id
  init pointer result;
```

Designated initializer.

```
dynamic
  components;
```

Retrieve all values from this result. The expected return type must fully match the actual return type.

```
dynamic
  component int n;
```

Retrieve the `n`th element from the result. The index of the first element is 0.

The expected return type must fully match, or can be an object, in which case numeric values are returned in a `Number` instance, and selectors in a `Selector`. Pointer values on a mismatch cause a condition to be signaled.

```
int
  length;
```

Return the number of elements in this result.

```
void
  setReturnValues (pointer, pointer) (first_value, extra_values)
      forSelector selector sel;
```

From the values held by this `InvocationResult`, update the `builtin_return_type` value pointed to by `first_value`, and any extra return value pointers as pointed to by the `va_list` `extra_values`.

```
TypeDescription
  typeDescription;
```

Return a the `TypeDescription` for this result.

```
OutputStream
  writeFields OutputStream s;
```

Output the elements in this result.

```
void
  dealloc;
```

Free the `values`.

```
void
  gc_mark_elements;
```

Mark the objects in this result.

```
void
  encodeUsingCoder Encoder coder;
```

Undocumented.

```
void
  initWithCoder Decoder coder;
```

Undocumented.

```
protected void
  encodeToCoder Encoder coder;
```

Private method to do the hard work of encoding all information carried by this result.

```
protected void
  decodeFromCoder Decoder coder;
```

Private method to do the hard work of decoding all information from the `coder` into this result.

File tom/Limits

class tom.Limits

After discussion on the TOM mailing list in April 2000, it was apparent that a <limits.h> equivalent is useful even when the values are the same on every platform.

variables

```
const BYTE_MIN = 0;
```

```
const BYTE_MAX = 255;
```

```
const CHAR_MIN = 0;
```

```
const CHAR_MAX = 65535;
```

```
const INT_MIN = -2147483648;
```

```
const INT_MAX = 2147483647;
```

```
const LONG_MIN = -9223372036854775808;
```

```
const LONG_MAX = 9223372036854775807;
```

instance tom.Limits

File tom/Lock

class tom.Lock

The abstract lock.

inherits

State supers: `State`

instance tom.Lock

methods

```
deferred void
  lock;
```

Undocumented.

```
deferred void
  unlock;
```

Undocumented.

```
deferred boolean
  tryLock;
```

Try to lock, and return `YES` if it succeeded.

```
Any (result)
  doWhileLocked Block block;
```

Lock, execute the `block`, and guarantee to unlock.

class tom.SimpleLock

A simple lock is a binary lock.

inherits

State supers: `Lock`

instance tom.SimpleLock

variables

```
pointer lock;
```

The underlying lock.

methods

```
void
  dealloc;
```

Undocumented.

```
id
  init;
```

Designated initializer.

```
void
  lock;
```

Undocumented.

```
void
  unlock;
```

Undocumented.

```
boolean
  tryLock;
```

Undocumented.

class tom.RecursiveLock

A recursive lock is a binary lock which can be obtained multiple times by the same thread.

inherits

State supers: `Lock`

instance tom.RecursiveLock

variables

```
pointer lock;
```

The underlying lock.

methods

```
void
  dealloc;
```

Undocumented.

```
id
  init;
```

Designated initializer.

```
void
  lock;
```

Undocumented.

```
void
```

```
  unlock;
```

Undocumented.

```
boolean
  tryLock;
```

Undocumented.

class tom.Semaphore

A semaphore is a lock which can be locked a number of times (1 for a binary semaphore) before the next attempt to lock will block.

When allocated first, the first `lock` will block.

inherits

State supers: `Lock`

methods

```
instance (id)
  new int num;
```

Return a new `Semaphore`, the first `num` `lock` operations will succeed.

instance tom.Semaphore

variables

```
pointer sem;
```

Pointer to the underlying structure.

methods

```
void
  dealloc;
```

Undocumented.

```
id
  init int num;
```

Designated initializer.

```
id
  init;
```

Another initializer.

```
void
  lock;
```

Undocumented.

```
void
  unlock;
```

Undocumented.

```
boolean
  tryLock;
```

Undocumented.

File tom/MutableArray

class tom.MutableArray

inherits

State supers: `Array`, `MutableIndexed`

methods

```
instance (id)
  withCapacity int cap;
```

Return a new instance of the receiving class which can hold `cap` elements without the need for resizing.

instance tom.MutableArray

variables

```
public int capacity;
```

The capacity of the array.

methods

```
void
  add All object;
```

Store the `object` at the end in the receiving array. If the receiving array stores unboxed values, the `object` is queried for its value.

```
(int, int)
```

```
  adjustMutableRange (int, int) (start, len);
```

Adjust the range (`start`, `len`) to fit the capacity of the receiving `MutableArray`. If `len` == -1, it is adjusted to fit the capacity.

```
void
  empty
post
  length == 0;
```

Empty the receiving array. This frees any storage used by the array to store its elements.

```
deferred id
  initWithCapacity int capacity;
```

Initialize the newly allocated receiving object to be able to hold `capacity` items without needing to resize.

```
id
  initWith int n
        at pointer addr;
```

Initialize with the indicated pointer and integer for contents and length. The `capacity` is set to the `length`.

```
deferred void
  insert All object
      at int index
pre
  index >= 0 && index <= length;
```

Insert the `object` at the `index`, shifting the objects at that or a higher index up by 1 position.

```
deferred Any
  removeAt int index;
```

Remove the element from `index`, decreasing the index of all elements after `index`, and return the element boxed. If the receiving array stores unboxed values, such as integers, the value returned is the element boxed.

```
void
  removeElementAt int index;
```

Remove the element from `index`, decreasing the index of all elements after `index`.

```
deferred void
  removeElements (int, int) (start, length);
```

Remove the `length` elements from `start`. If `length` == -1, all elements from `start` are removed.

```
void
  resize int to
pre
  to >= 0;
```

Adjust the size of the array, filling any newly created entries with the default value for the type (i.e. 0).

```
deferred void
  resize (int, int) (start, num);
```

Adjust the size of the array by inserting `num` new entries at `start` filling newly created entries with the default values for the type (i.e. 0).

```
void
  truncate int new_length
pre
  new_length >= 0;
```

Adjust the length of this array to `new_length`.

```
void
  bubbleSortUsingKey selector key
         comparator: selector cmp = selector (int compare All);
```

Bublesort the receiving array on the `key` of the contained elements by comparing them using the `compare` selector.

```
void
  quickSortUsingKey selector key
        comparator: selector compare = selector (int compare All);
```

Quicksort the receiving array on the `key` of the contained elements by comparing them using the `compare` selector.

```
id
  initCopy;
```

In addition to what our `super` does, adjust our (new) `capacity` to fit our `length`.

File tom/MutableByteArray

class tom.MutableByteArray

inherits

State supers: `ByteArray`, `MutableArray`, `OutputStream`

instance tom.MutableByteArray

methods

```
void
  add byte b;
```

Undocumented.

```
void
  freeze;
```

Undocumented.

```
id
  initWithCapacity int capacity;
```

Undocumented.

```
protected id
  initWithEnumerator Enumerator e;
```

Undocumented.

```
byte
  removeAt int index;
```

Undocumented.

```
Any
  removeAt int index;
```

Undocumented.

```
void
  removeElements (int, int) (start, length);
```

Undocumented.

```
void
  resize (int, int) (start, num);
```

Undocumented.

```
void
  set byte b
   at int index;
```

Undocumented.

```
void
```

```
  set char c
   at int index;
```

Undocumented.

```
void
  set All object
   at int index;
```

Undocumented.

```
void
  swap (int, int) (i, j);
```

Undocumented.

```
void
  initWithCoder Decoder coder;
```

Undocumented.

```
void
  close;
```

Undocumented.

```
int
       readRange (int, int) (start, num)
  fromByteStream ByteStream f;
```

Undocumented.

```
int
      readRange (int, int) (start, num)
  fromByteArray ByteArray source
             to int position;
```

Undocumented.

```
void
  write byte b;
```

Undocumented.

```
int
  write byte b;
```

Undocumented.

```
int
  writeBytes int num
```

```
        from pointer address;
```

Undocumented.

```
int
  writeBytes int num
        from pointer address
          at int offset;
```

Undocumented.

class tom.Data

inherits

State supers: `MutableByteArray`

instance tom.Data

File tom/MutableByteString

class tom.MutableByteString

inherits

State supers: `ByteString`, `MutableString`, `MutableByteArray`

instance tom.MutableByteString

methods

```
void
  freeze;
```

Undocumented.

```
ByteString
  frozen;
```

Undocumented.

```
protected id
  init (pointer, int) (p, num);
```

Undocumented.

```
id
```

```
  initCopy (pointer, int) (p, num);
```

In addition to what our `super` does, adjust our (new) `capacity` to fit our `length`.

```
void
  set char c
   at int index;
```

Set the byte at `index` to the character `c`, converted to a `byte` according to the `default_encoding`.

```
void
  add char c;
```

Add the byte encoding of the `char c` to this string.

```
ByteString
  substring (int, int) (start, len);
```

Override the `ByteString` implementation of this `substring` method, since that actually employs `ByteSubstring` objects which we can't use.

File tom/MutableCharArray

class tom.MutableCharArray

inherits

State supers: `CharArray`, `MutableArray`

instance tom.MutableCharArray

methods

```
void
  add char c;
```

Undocumented.

```
void
  freeze;
```

Undocumented.

```
id
  initWithCapacity int capacity;
```

Undocumented.

```
protected id
  initWithEnumerator Enumerator e;
```

Undocumented.

```
char
  removeAt int index;
```

Undocumented.

```
void
  removeElements (int, int) (start, length);
```

Undocumented.

```
void
  resize (int, int) (start, num);
```

Undocumented.

```
void
  set char c
   at int index;
```

Undocumented.

```
CharNumber
  removeAt int index;
```

Undocumented.

```
void
  set All object
   at int index;
```

Undocumented.

```
void
  swap (int, int) (i, j);
```

Undocumented.

File tom/MutableCharString

class tom.MutableCharString

inherits

State supers: `CharString`, `MutableString`, `MutableCharArray`

instance tom.MutableCharString

methods

```
void
  freeze;
```

Undocumented.

```
protected id
  init (pointer, int) (p, num);
```

Undocumented.

File tom/MutableDoubleArray

class tom.MutableDoubleArray

inherits

State supers: `DoubleArray`, `MutableArray`

instance tom.MutableDoubleArray

methods

```
void
  add double d;
```

Undocumented.

```
void
  freeze;
```

Undocumented.

```
id
  initWithCapacity int capacity;
```

Undocumented.

```
protected id
  initWithEnumerator Enumerator e;
```

Undocumented.

```
double
  removeAt int index;
```

Undocumented.

```
Any
  removeAt int index;
```

Undocumented.

```
void
  removeElements (int, int) (start, length);
```

Undocumented.

```
void
  resize (int, int) (start, num);
```

Undocumented.

```
void
  set double d
   at int index;
```

Undocumented.

```
void
  set All object
   at int index;
```

Undocumented.

```
void
  swap (int, int) (i, j);
```

Undocumented.

File tom/MutableFloatArray

class tom.MutableFloatArray

inherits

State supers: `FloatArray`, `MutableArray`

instance tom.MutableFloatArray

methods

```
void
  add float f;
```

Undocumented.

```
void
  freeze;
```

Undocumented.

```
id
  initWithCapacity int capacity;
```

Undocumented.

```
protected id
  initWithEnumerator Enumerator e;
```

Undocumented.

```
float
  removeAt int index;
```

Undocumented.

```
Any
  removeAt int index;
```

Undocumented.

```
void
  removeElements (int, int) (start, length);
```

Undocumented.

```
void
  resize (int, int) (start, num);
```

Undocumented.

```
void
  set float f
   at int index;
```

Undocumented.

```
void
  set All object
   at int index;
```

Undocumented.

```
void
  swap (int, int) (i, j);
```

Undocumented.

File tom/MutableIntArray

class tom.MutableIntArray

inherits

State supers: `IntArray`, `MutableArray`

instance tom.MutableIntArray

methods

```
void
  add int i;
```

Undocumented.

```
void
  freeze;
```

Undocumented.

```
id
  initWithCapacity int capacity;
```

Undocumented.

```
protected id
  initWithEnumerator Enumerator e;
```

Undocumented.

```
int
  removeAt int index;
```

Undocumented.

```
Any
  removeAt int index;
```

Undocumented.

```
void
```

```
  removeElements (int, int) (start, length);
```

Undocumented.

```
void
  resize (int, int) (start, num);
```

Undocumented.

```
void
  set int i
   at int index;
```

Undocumented.

```
void
  set All object
   at int index;
```

Undocumented.

```
void
  swap (int, int) (i, j);
```

Undocumented.

File tom/MutableObjectArray

class tom.MutableObjectArray

inherits

State supers: `ObjectArray`, `MutableArray`

methods

```
instance (id)
    with int num
  copies All o;
```

Return a newly allocated `MutableObjectArray` with `num` copies of the object reference `o`. Since `o` is a reference to an actual object, only `num` copies of that reference are stored: the object is not copied at all.

instance tom.MutableObjectArray

methods

```
void
  freeze;
```

Undocumented.

```
id
  initWithCapacity int capacity;
```

Undocumented.

```
protected id
  initWithEnumerator Enumerator e;
```

Undocumented.

```
void
  insert All object
      at int index;
```

Undocumented.

```
Any
  removeAt int index;
```

Undocumented.

```
void
  removeElements (int, int) (start, length);
```

Undocumented.

```
void
  resize (int, int) (start, num);
```

Undocumented.

```
void
  set All object
   at int index;
```

Undocumented.

```
void
  swap (int, int) (i, j);
```

Undocumented.

```
void
  encodeUsingCoder Encoder coder;
```

Undocumented.

```
void
  initWithCoder Decoder coder;
```

Undocumented.

File tom/MutablePointerArray

class tom.MutablePointerArray

inherits

State supers: `PointerArray`, `MutableArray`

instance tom.MutablePointerArray

methods

```
void
  add pointer d;
```

Undocumented.

```
void
  freeze;
```

Undocumented.

```
id
  initWithCapacity int capacity;
```

Undocumented.

```
protected id
  initWithEnumerator Enumerator e;
```

Undocumented.

```
pointer
  removeAt int index;
```

Undocumented.

```
Any
  removeAt int index;
```

Undocumented.

```
void
```

```
  removeElements (int, int) (start, length);
```

Undocumented.

```
void
  resize (int, int) (start, num);
```

Undocumented.

```
void
  set pointer d
   at int index;
```

Undocumented.

```
void
  set All object
   at int index;
```

Undocumented.

```
void
  swap (int, int) (i, j);
```

Undocumented.

File tom/MutableString

class tom.MutableString

inherits

State supers: `String`, `MutableArray`

instance tom.MutableString

methods

```
deferred void
  set char c
   at int i;
```

Undocumented.

File tom/Number

class tom.Number

inherits

State supers: `State`

methods

```
instance (id)
  with byte value;
```

Undocumented.

```
instance (id)
  with char value;
```

Undocumented.

```
instance (id)
  with int value;
```

Undocumented.

```
instance (id)
  with long value;
```

Undocumented.

```
instance (id)
  with float value;
```

Undocumented.

```
instance (id)
  with double value;
```

Undocumented.

instance tom.Number

methods

```
deferred byte
  byteValue;
```

Undocumented.

```
deferred char
  charValue;
```

Undocumented.

```
deferred int
  intValue;
```

Undocumented.

```
deferred long
  longValue;
```

Undocumented.

```
deferred float
  floatValue;
```

Undocumented.

```
deferred double
  doubleValue;
```

Undocumented.

```
int
  hash;
```

Undocumented.

```
boolean
  equal Number n;
```

Undocumented.

```
deferred int
  compare Number n;
```

Undocumented.

```
deferred int
  compare byte v;
```

Undocumented.

```
deferred int
  compare char v;
```

Undocumented.

```
deferred int
  compare int v;
```

Undocumented.

```
deferred int
  compare long v;
```

Undocumented.

```
deferred int
  compare float v;
```

Undocumented.

```
deferred int
  compare double v;
```

Undocumented.

```
deferred protected id
  init byte value;
```

Undocumented.

```
deferred protected id
  init char value;
```

Undocumented.

```
deferred protected id
  init int value;
```

Undocumented.

```
deferred protected id
  init long value;
```

Undocumented.

```
deferred protected id
  init float value;
```

Undocumented.

```
deferred protected id
  init double value;
```

Undocumented.

```
boolean
  dump_simple_p;
```

Undocumented.

File tom/ObjectArray

class tom.ObjectArray

inherits

State supers: `Array`

methods

```
instance (id)
  with dynamic elements;
```

Undocumented.

instance tom.ObjectArray

methods

```
protected id
  initWithEnumerator Enumerator e;
```

Undocumented.

```
int (v)
  hash;
```

Hash some elements.

```
boolean
  equal id other;
```

Compare the elements.

```
Any
  at int index
pre
  index >= 0 && index < length
post
  length == old length;
```

Return the object at `index` in the receiving array.

```
boolean
  dump_self_p;
```

Return `TRUE`.

```
void
  dumpSelf MutableKeyed done
```

```
    indent MutableByteString prefix
    simple boolean allow_simple
     level int level
        to OutputStream s;
```

Dump the elements to the stream `s`.

```
int
  elementByteSize;
```

Undocumented.

```
(pointer, int)
  pointerToElements (int, int) (start, len);
```

Undocumented.

```
protected void
  setDuringConstruction (int, All) (index, object)
pre
  index >= 0 && index < length;
```

Set the `object` at the `index` in the receiving array, even if it is not a mutable array. This method /must/ only be used during construction of a constant array object.

```
class (State)
  mutableCopyClass;
```

Return the `MutableObjectArray` class.

```
id
    deepen int level
  mutably: boolean mutable_p = NO;
```

Deepen this copy.

```
void
  encodeUsingCoder Encoder coder;
```

Undocumented.

```
void
  initWithCoder Decoder coder;
```

Undocumented.

```
redefine void
  gc_mark_elements;
```

This method is invoked by the garbage collector to have the receiving object mark the elements it references. An `ObjectArray` must reference the objects it holds.

File tom/Pointer

class tom.Pointer

This `Pointer` class is a simple object wrapper around an object value.

inherits

State supers: `State`

methods

```
int (result)
  hash pointer p;
```

Hash the pointer `p`. This hashes the pointer in a way similar to the `hashq` method of `All`.

```
instance (id)
  with pointer p;
```

Simple allocator.

instance tom.Pointer

variables

```
public pointer value;
```

Our value.

methods

```
id (self)
  init pointer p;
```

Designated initializer.

```
boolean
  dump_simple_p;
```

Return `YES`.

```
boolean
  equal Pointer other;
```

Compare our `value` with the `other`'s.

```
int
  hash;
```

Return the hashed pointer `value`.

```
OutputStream
  write OutputStream s;
```

Similar to a `Number`, a `Pointer` simply outputs the `value`.

File tom/PointerArray

class tom.PointerArray

inherits

State supers: `Array`

methods

```
instance (id)
  with dynamic elements;
```

Undocumented.

instance tom.PointerArray

methods

```
protected id
  initWithEnumerator Enumerator e;
```

Undocumented.

```
Any
  at int index;
```

Undocumented.

```
pointer
  at int index;
```

Undocumented.

```
int
  elementByteSize;
```

Undocumented.

```
(pointer, int)
  pointerToElements (int, int) (start, len);
```

Undocumented.

```
class (State)
  mutableCopyClass;
```

Return the `MutablePointerArray` class.

File tom/PointerDictionary

class tom.PointerDictionary

A `PointerDictionary` maps a `pointer` to an object reference.

inherits

State supers: `EqHashTable`, `DictionaryContainer`

instance tom.PointerDictionary

methods

```
Any
  at pointer key;
```

Undocumented.

class tom.MutablePointerDictionary

inherits

State supers: `PointerDictionary`, `MutableEqHashTable`

instance tom.MutablePointerDictionary

methods

```
void
  remove pointer key;
```

Remove the mapping for the `key`.

```
void
  set All value
   at pointer key
pre
```

```
  value != nil;
```

Associate the `value` with the `key`.

File tom/Queue

class tom.Queue

The abstract Queue class. There is not much you can do with it as queue has to be mutable to do anything useful.

inherits

State supers: `Indexed`

instance tom.Queue

methods

```
boolean
  queuep;
```

Return TRUE.

```
deferred boolean
  emptyp;
```

Return TRUE for empty queues.

class tom.MutableQueue

inherits

State supers: `Queue`, `MutableIndexed`

instance tom.MutableQueue

methods

```
deferred void
  add Any object;
```

Push an object to the head of the queue.

```
deferred Any
  pop;
```

Pop the object from the tail of the queue.

class tom.MutableObjectQueue

inherits

State supers: `MutableQueue`, `Conditions`

instance tom.MutableObjectQueue

variables

```
int offset;
```

The index into the `contents` of the element `self[0]`.

```
int elements;
```

The number of elements.

```
MutableObjectArray contents;
```

The array holding the actual objects.

methods

```
Any
  at int index
pre
  index >= 0 && index < elements;
```

Undocumented.

```
void
  set All object
   at int index
pre
  index >= 0 && index < elements;
```

Undocumented.

```
int
  length;
```

Undocumented.

```
void
  add Any object;
```

Add the `object` to the end.

```
boolean
  emptyp;
```

Undocumented.

```
Any
  pop;
```

Retrieve the object at `self[0]` and remove it. All in constant time, of course.

```
id (self)
  empty;
```

Make the queue empty.

File tom/Random

class tom.Random

inherits

Behaviour supers: `All`

instance tom.Random

inherits

Behaviour supers: `All`

methods

```
deferred double
  next;
```

Return a double random number.

```
deferred int
  next;
```

Return an integer random number.

```
int (result)
  next int limit
pre
  limit > 1
post
  result >= 0 && result < limit;
```

Return a number in the range [0, limit).

class tom.IntegerRandom

Superclass for use by random number generators which customarily return ints.

inherits

State supers: `Random`, `Limits`

instance tom.IntegerRandom

methods

```
double
  next;
```

Return the next double value, in the range [0, 1.0).

class tom.DoubleRandom

Superclass for use by random number generators which customarily return ints.

inherits

State supers: `Random`, `Limits`

instance tom.DoubleRandom

methods

```
int (result)
  next;
```

Return the next integer value, in the range [0, INT_MAX].

class tom.MinimalRandom

MinimalRandom implements `IntegerRandom`. It uses seeds to generate a repeatable sequence of pseudo-random integers.

inherits

State supers: `State`, `IntegerRandom`

variables

```
const ia = 16807;
```

Constants needed by the algorithm.

```
const im = 2147483647;
```

```
const iq = 12773;
```

```
const ir = 2836;
```

instance tom.MinimalRandom

variables

```
int seed;
```

The seed from which we feed.

methods

```
id (self)
  init int initial_seed
pre
  initial_seed > 0;
```

Designated initializer.

```
id
  init;
```

Initialize this instance with a seed derived from the current moment in time and the `hashq` value of `self`.

```
int
  next;
```

Return the next random positive `int` value.

File tom/RandomDouble

class tom.Ranecu

This is a Random number generator called Ranecu. A lot of the actual code was borrowed from the RngPack 1.0 Java package by Paul Houle.

This is a quote from the "Ranecu.java":

"Ranecu is an advanced multiplicative linear congruential random number generator with a period of aproximately 10e18"

References:

http://www.msc.cornell.edu/~houle/rngpack

F. James, "Comp. Phys. Comm." 60 1990 p 329-344 P. L‘Ecuyer, "Commun. ACM." 1988 1988 p 742.

inherits

State supers: `State`, `DoubleRandom`

variables

```
const DEFSEED1 = 12345;
```

Default seeds.

```
const DEFSEED2 = 67890;
```

methods

```
instance (id) (r)
  new;
```

Return a newly allocated instance, initialized with a random seed.

```
instance (id)
  newWithDefault;
```

Return a newly allocated instance, initialized with a default seed.

```
instance (id)
  newWithSeed int seed;
```

Return a newly allocated instance, initialized with the specified `seed`.

instance tom.Ranecu

variables

```
int iseed1;
```

The seed.

```
int iseed2;
```

methods

```
id (self)
  initWithSeed (int, int) (s1, s2);
```

Designated initializer.

```
id
  init;
```

Default initializer.

```
id
  initWithSeed int seed;
```

Short initializer.

```
id
  initWith Date d;
```

Initialize with the date `d`.

```
double
  next;
```

Undocumented.

```
double
  raw;
```

Undocumented.

```
long
  getSeed;
```

Undocumented.

class tom.Ranlux

This is a Random number generator called Ranlux. A lot of the actual code was borrowed from the RngPack 1.0 Java package by Paul Houle.

References:

http://www.msc.cornell.edu/~houle/rngpack F. James, "Computer Physics Communications" 79 (1994) 111 http://www.camk.edu.pl/~tomek/html.refs/ranlux.f90_2.html.

inherits

State supers: `State`, `DoubleRandom`

variables

```
const maxlev = 4;
```

Maximum luxury level.

```
const lxdflt = 3;
```

Default luxury level.

```
const igiga = 1000000000;
```

```
const jsdflt = 314159265;
```

```
const twop12 = 4096;
```

```
const itwo24 = 1 << 24;
```

```
const icons = 2147483563;
```

methods

```
instance (id) (r)
  new;
```

Undocumented.

```
instance (id)
  newWithDefault;
```

Undocumented.

```
instance (id)
  newWithSeed int seed;
```

Undocumented.

instance tom.Ranlux

variables

```
MutableIntArray iseeds;
```

```
MutableIntArray isdext;
```

```
MutableIntArray next;
int luxlev;
int nskip;
int inseed;
int jseed;
int in24;
int kount;
int mkount;
int i24;
int j24;
MutableFloatArray seeds;
float carry;
float twom24;
float twom12;
```

```
MutableIntArray ndskip;
```

methods

```
id (self)
  init;
```

Undocumented.

```
id (self)
  initWithSeed int ins;
```

Undocumented.

```
id (self)
  initWithSeed int ins
         atLux int lux;
```

Undocumented.

```
void
  initLux;
```

Undocumented.

```
void
  initArrays;
```

Undocumented.

```
double
  next;
```

Undocumented.

```
double
  raw;
```

Undocumented.

```
void
  rluxdef;
```

Undocumented.

```
void
  rluxgo (int, int) (lux, ins);
```

Undocumented.

class tom.Ranmar

This is a Random number generator called Ranmar. A lot of the actual code was borrowed from the RngPack 1.0 Java package by Paul Houle.

This is a quote from the "Ranmar.java" :

"[Ranmar] is a lagged Fibonacci generator proposed by Marsaglia and Zaman and is a good research grade generator."

References:

http://www.msc.cornell.edu/~houle/rngpack.

inherits

State supers: `DoubleRandom`, `State`

variables

```
const DEFSEED = 54217137;
```

Default seed.

```
const BIG_PRIME = 899999963;
```

The 46,009,220th prime number, the largest prime less than 9*10e8. Used as a modulus because this version of `RANMAR` needs a seed between 0 and 9*10e8 and `BIG_PRIME` isn't commensurate with any regular period.

methods

```
instance (id) (r)
  new;
```

Undocumented.

```
instance (id)
  newWithDefault;
```

Undocumented.

```
instance (id)
  newWithSeed int seed;
```

Undocumented.

instance tom.Ranmar

variables

```
MutableDoubleArray u;
```

```
MutableDoubleArray uvec;
```

```
double c;
```

```
double cd;
```

```
double cm;
```

```
int i97;
```

```
int j97;
```

methods

```
id (self)
  init;
```

Undocumented.

```
id (self)
  initWithSeed int seed;
```

Undocumented.

```
id (self)
  initWith Date d;
```

Undocumented.

```
void
  ranmarin int ijkl;
```

Undocumented.

```
double
  next;
```

Undocumented.

```
double
  raw;
```

Undocumented.

File tom/Runtime

class tom.Runtime

The Runtime class provides an interface to the functionality in the runtime library and other process related information.

Most variables of the `Runtime` class are not `public`, since they can be accessed by simply inheriting `Runtime`.

inherits

State supers: `Conditions`, `Constants`, `stdio`

variables

```
static String hostname;
```

The name of the host on which this program is running.

```
static public String program_name;
```

The name under which this program was invoked (i.e. the basename of argv[0] in C).

```
static public String long_program_name;
```

The long program name (i.e. argv[0] in C).

```
static Array arguments;
```

The arguments as passed to `main`.

```
static Array all_arguments;
```

All the arguments, before any `load` method modified it.

```
static Mapped environment;
```

The environment.

```
static private ByteString main_resource_dir;
```

The directory holding the main resources, at least including the character encodings.

`static private MutableMapped classes_by_name;`

The dictionary of classes. Mapping from name to array of classes with that name. Since this is created upon request and it is reset by dynamic loading, it is not publicly available. Access it through the `classes_by_name` method.

`static private int quit_inhibit;`

Iff !0, quit (`SIGINT`, the user interrupt signal) is inhibited.

`static boolean quit_pending;`

Iff !0, a `signal-int` will be raised when `quit_inhibit` and `panic_mode` again reach 0.

`static int panic_mode;`

Iff !0, any signal received (excluding the interrupt signal), other condition signaled or raised, or any object thrown will simply cause an abort. This is used to protect critical sections in the runtime, such as during garbage collection or object allocation. If `panic_mode` is set, `quit_inhibit` is implicitly set too.

`static boolean core_on_fatal;`

Iff `TRUE`, a corefile will be produced on fatal errors, such as uncaught condition raises.

`static boolean stacktrace_on_fatal;`

Iff `TRUE`, a stacktrace will be printed on fatal errors, such as uncaught condition raises. This facility is dependent upon the stacktrace generation being implemented on the platform in use.

`static int gc_alloc_since_partial;`

The number of objects allocated since the last partial garbage collection run. In this respect, a partial run completing a full run is still considered a partial run.

`static int gc_alloc_since_total;`

The number of objects allocated since the previous completed run. This excludes the objects counted by `gc_alloc_since_partial`; it is adjusted after a run is initiated, before the run is actually started.

`static boolean gc_stat_at_exit;`

Iff `TRUE` statistics on memory usage and the garbage collector will be emitted upon exit.

`static boolean rt_stat_at_exit;`

Iff `TRUE` output statistics on the runtime structures at exit.

`static boolean rt_num_inst_at_exit;`

Iff `TRUE` output the number of live instances of each class at exit.

`static int gc_num_runs;`

Statistics on the garbage collector and allocator, in order: the number of gc runs; the number of runs which complete a full run; the number of object allocated; the number of objects deallocated; the (real, i.e. elapsed) time spent protecting, marking, and sweeping; and the time spent in all of gc (this is the sum of the previous three, plus overhead).

`static int gc_num_complete;`

`static long gc_num_alloc;`

`static long gc_num_dealloc;`

`static double gc_total_protect;`

`static double gc_total_mark;`

`static double gc_total_sweep;`

`static double gc_total_all;`

`static int malloc_cur_bytes;`

These numbers are only maintained if the runtime library was not instructed to not do so at compile time.

`static int malloc_max_bytes;`

`static int malloc_cum_bytes;`

`static int malloc_cur_items;`

`static int malloc_max_items;`

`static int malloc_cum_items;`

`static int gc_inhibit;`

Iff !0, garbage collection won't be run. This is important during, for instance, enumerating a Container, since (most) enumerators can not handle the collection changing while they are enumerating.

`static private boolean gc_atomic;`

Iff `TRUE` (the default), garbage collection will run atomically, irrespective of the time constraint argument to `garbageCollect`. When running with atomic garbage collection, new objects are white (presumed dead) whereas with non-atomic garbage collection, new objects are gray (presumably alive).

A program using atomic garbage collection needs less memory, since only one run is needed to reclaim a dead object, instead of two runs. It also means that, for example, in a multi-threaded program, the thread doing garbage collection will block all other threads.

`static boolean gc_atomic_next;`

The desired value of `gc_atomic`, which will take effect after the next GC run. Default is whatever the value of `gc_atomic` was at startup.

`static boolean gc_full_at_exit;`

Iff `TRUE` all garbage will be cleaned upon exit. This is a debugging tool mostly.

`static int gc_debug;`

The level of debugging garbage output by the garbage collector. Information is output to `stderr` stream provided by the C library. No information will be output if `gc_debug` is 0 or if the runtime was not compiled with the appropriate flags.

`static int gc_partial_threshold;`

Threshold for `gc_alloc_since_partial` before a garbage collection run will be initiated. If `gc_partial_threshold` is 0, garbage collection is never run implicitly. The default value is 25000, or the value passed as `:gc-pth` on the command line.

`static double gc_partial_time_limit;`

The time allowed for a partial garbage collection run when initiated by `gc_alloc_since_partial` exceeding `gc_partial_threshold`. The default is 0, implying no time limit.

`static int gc_total_threshold;`

When a partial garbage collection run is initiated and `gc_alloc_since_total` exceeds `gc_total_threshold`, the `gc_partial_time_limit` is ignored and instead the `gc_total_time_limit` is used. If `gc_total_threshold` is 0, it is ignored.

```
static double gc_total_time_limit;
```

The time limit used in case the condition described for `gc_total_threshold` applies.

```
static boolean preconditions_enabled;
```

Iff `TRUE`, preconditions are checked.

```
static boolean postconditions_enabled;
```

Iff `TRUE`, postconditions are checked.

```
static boolean rt_print_signals;
```

Iff `TRUE`, unhandled signals are printed on `[stdio err]`. This is for debugging purposes.

methods

```
int
      start (All, selector) (object, sel)
  arguments Array arguments;
```

This method is invoked by the runtime library. Its main responsibility is to invoke the real main method, which is identified by the `sel` and `object`.

```
void
  exit int rc;
```

Normal level exit. Cleaning up will be performed.

```
void
  fastExit int rc;
```

Low level exit. Usual functionality for cleaning up is avoided.

```
void
  unhandledSignal Condition condition;
```

Output information on the unhandled signal `condition` on `[stdio err]`.

```
void
  willExit int rc;
```

Perform all things necessary for a clean exit. This runs the garbage collector if specified by a `:gc-exit`, dumps gc statistics if specified by `:gc-stat`, number of instances if specified by `:rt-inst`, and memory overhead information if specified by `:rt-stat`.

```
OutputStream
  help OutputStream s
  done MutableKeyed done;
```

Output help information about the facilities (most notably ':' arguments) offered by the receiving class, on the `OutputStream` `s`.

Any implementation should add itself to the set `done`, and check for presence before outputting anything, to avoid generating the same output for every subclass not overriding this method.

```
void
  load MutableArray arguments;
```

Scan the arguments to the program for something telling us whether or how to do certain things.

See the output of `:help` of any TOM program for short information on the options.

```
void
  preload MutableArray arguments;
```

Invoked by the runtime library before the first load is invoked.

This method is needed for two occasions: first is to check for `:help`. The reason this is not done in `load` is to be able to get some help before any negative side effects of any `load` method. The second reason is for finding `:rt-resource-dir`, which must be done before `ByteString`'s `load` method can play with its encoding.

```
void
  reportNumInstances OutputStream s
      includeZeroes: boolean zeroes = FALSE;
```

Output to the stream `s` the number of live instances for each class. If the optional `zeroes` is `TRUE`, classes with zero instances are included in the report. This includes deferred classes, as they cannot have any instances.

```
protected void
  runtimeStatistics OutputStream s;
```

Output the actual statistics for `:rt-stat` to the stream `s`.

```
Indexed
  classes;
```

Return the array of all class objects.

```
Mapped
  classes_by_name;
```

Return the, possibly created upon request, mapped collection of classes keyed on their name.

```
class (State) (class_object)
  classNamed String name;
```

Return the class with the `name`.

Name may be unqualified, as in `"Runtime"`, which will return the single class with that name, or `nil` in case such a class does not exist, or if more than one class with that name exists in multiple units.

The `name` may be qualified, as in `"tom.Runtime"`, in which case the `Runtime` class of the `tom` unit will be returned, if that unit and class within that unit exist.

```
selector
  selectorNamed String name;
```

Return the existing selector known by the `name`.

```
boolean
  selector selector s1
    equals selector s2;
```

Return `TRUE` iff the selectors `s1` and `s2` denote the same selector.

```
String
  nameOfSelector selector sel;
```

Return the name of the `selector`.

```
selector
  nullSelector;
```

Return the invalid selector.

```
void
  garbageCollect;
```

Run the garbage collector to the end of a full garbage collection run.

```
void
  garbageCollect double time
pre
  gc_atomic -> !time;
```

Run the garbage collector for at most `time` seconds.

```
void
  disableGC
pre
  gc_inhibit >= 0
post
  gc_inhibit == old gc_inhibit + 1;
```

Increase the `gc_inhibit`. This invocation should be matched by an invocation of `enableGC`.

```
void
  enableGC
pre
```

```
  gc_inhibit > 0
post
  gc_inhibit == old gc_inhibit - 1;
```

Decrease the `gc_inhibit`.

```
Mapped
  environment;
```

Return the dictionary holding the process environment. The dictionary is filled upon the first request, thread-safely.

```
ByteString
  main_resource_dir;
```

Return the `main_resource_dir`.

```
void
  setenv (String, String) (environment_variable, value);
```

Set the `value` of the `environment_variable`, thread-safely.

```
String
  hostname;
```

Return the `hostname` of this machine. If the class variable is not set, it is set once from gethostname(2).

```
String
  tom_prefix;
```

Return the directory in which all TOM stuff has been installed. This returns the value of `TOM_PREFIX` in the `Constants` class.

```
Any
  perror String prefix
     for All object
   class ConditionClass condition_class
   raise boolean not_signal;
```

Construct a `Condition` for the `object` with the `condition_class` and a message created from the (optional) prefix, plus the information available from the (ANSI C) `errno` variable. If `not_signal` is `TRUE`, the new condition is raised; otherwise it is signaled and the result is returned (if a return is allowed).

```
int
  quit_inhibit;
```

Accessor method for `quit_inhibit` which is private to the `Runtime` class to protect it against being mutated by subclasses but which can be freely read, hence this method.

```
void
  quit_disable;
```

Increase the quit_inhibit flag. Any increase should be accompanied later on by the corresponding decrease.

```
void
  quit_enable
pre
  quit_inhibit > 0;
```

Decrease the quit_inhibit flag, raising a postponsed signal-int if indicated by quit_pending.

```
int
  panic_mode;
```

Accessor method for panic_mode which is private to the Runtime class to protect it against being mutated by subclasses but which can be freely read, hence this method.

```
void
  panic_enable;
```

Increase the panic_mode flag. Any increase should be accompanied later on by the corresponding decrease.

```
void
  panic_disable
pre
  panic_mode > 0;
```

Decrease the panic_mode flag, raising a signal-int if requested by quit_pending.

```
Array
  crawlStack;
```

Return the return addresses currently outstanding on the CPU stack (up to the first 100). Information about these pointers may be obtained from [Runtime symbolInfo].

```
(ByteString, pointer, ByteString, pointer, pointer) (file_name, base_address, sym-
bol_name, symbol_address, offset)
  symbolInfo pointer address;
```

Return extensive symbol information on the ADDRESS.

```
void
    printStack OutputStream stream
  ignoreUntil: String ignore_until_symbol = "";
```

Print a stack trace to an OutputStream.

instance tom.Runtime

The `Runtime` instance is totally empty.

File tom/Selector

class tom.Selector

A `Selector` is an object wrapper for a selector.

inherits

State supers: `State`

methods

```
instance (id)
  with selector sel;
```

Return an instance of `Selector` wrapping the `selector` `sel`.

instance tom.Selector

variables

```
selector sel;
```

The selector which we wrap.

methods

```
id (self)
  init selector s;
```

Designated initializer.

```
int
  hash;
```

Return the identity of the selector.

```
boolean
  equal id other;
```

Return whether or not the selector wrapped by `self` is equal to `other`.

```
String
  name;
```

Return the name of the selector.

```
selector
  selector;
```

Return the selector we wrap.

```
void
  encodeUsingCoder Encoder coder;
```

Encode the receiving `Selector` object. This writes the name of the selector as the means of identifying the selector upon decoding. (The selector itself can't be written since encoding a selector actually encodes a `Selector` object.)

```
void
  initWithCoder Decoder coder;
```

Initialize the receiving `Selector` from the `coder`.

File tom/Set

class tom.Set

inherits

State supers: `HashTable`, `Keyed`

instance tom.Set

methods

```
Any
  member All object;
```

Invoke `HashTable`'s implementation.

```
Any
  memq All object;
```

Invoke `HashTable`'s implementation.

```
Enumerator
  enumerator;
```

Return an enumerator on the receiving set.

```
Enumerator
  keyEnumerator;
```

Undocumented.

class tom.MutableSet

inherits

State supers: `Set`, `MutableHashTable`, `MutableKeyed`, `Container`

instance tom.MutableSet

methods

```
void
  add All object;
```

Undocumented.

```
void
  remove All object;
```

Remove `elt` from the receiving set, if present.

class tom.SetEnumerator

inherits

State supers: `HashTableEnumerator`

instance tom.SetEnumerator

variables

```
redeclare BucketSetElement elt;
```

methods

```
(boolean, Any) (valid, object)
  next;
```

Undocumented.

File tom/Sorted

class tom.SortedKeyed

The SortedKeyed class keeps its elements in ascending order.

inherits

State supers: `Keyed`

instance tom.SortedKeyed

methods

```
deferred Enumerator
  between (Comparable, Comparable) (start, last);
```

Enumerate the elements in an interval. `nil` at the either end signifies the first or the last element.

```
deferred Any
  lowest;
```

Undocumented.

```
deferred Any
  highest;
```

Undocumented.

```
deferred redeclare Any
  at Comparable object;
```

Undocumented.

```
deferred redeclare Any
  member Comparable object;
```

Undocumented.

```
deferred redeclare Any
  memq Comparable object;
```

Undocumented.

class tom.MutableSortedKeyed

inherits

State supers: `SortedKeyed`, `MutableKeyed`

instance tom.MutableSortedKeyed

methods

```
redeclare void
```

```
  add Comparable object;
```

Undocumented.

```
redeclare void
  remove Comparable object;
```

Undocumented.

class tom.SortedMapped

inherits

State supers: `Mapped`, `SortedKeyed`

instance tom.SortedMapped

methods

```
deferred Enumerator
  valuesOfKeysBetween (Comparable, Comparable) (start, last)
         includeLeft: boolean incleft = TRUE
        includeRight: boolean incright = TRUE;
```

Enumerate the values in an interval.

class tom.MutableSortedMapped

inherits

State supers: `MutableSortedKeyed`, `MutableMapped`

instance tom.MutableSortedMapped

methods

```
deferred redeclare void
  set All value
   at Comparable key;
```

Undocumented.

class tom.SortedObjectArray

inherits

State supers: `SortedKeyed`

instance tom.SortedObjectArray

variables

```
public MutableObjectArray contents;
```

The array we employ to actually store the contents.

methods

```
boolean (result)
  verifySortedContents;
```

Undocumented.

```
id
  initWithSortedEnumerator Enumerator e
post
  [self verifySortedContents];
```

Undocumented.

```
id
  initWithEnumerator Enumerator e;
```

Undocumented.

```
ObjectArray
  allKeys;
```

Undocumented.

```
(boolean, int)
  indexOf Comparable object;
```

The guts of the binary search algorithm.

```
Any
  at Comparable object;
```

Undocumented.

```
Enumerator
  valuesOfKeysBetween (Comparable, Comparable) (start, last)
         includeLeft: boolean incleft = TRUE
        includeRight: boolean incright = TRUE;
```

Undocumented.

```
Any
  lowest
```

```
pre
  [contents length] != 0;
```

Undocumented.

```
Any
  highest
pre
  [contents length] != 0;
```

Undocumented.

class tom.MutableSortedObjectArray

inherits

State supers: `MutableSortedKeyed`, `SortedObjectArray`

instance tom.MutableSortedObjectArray

methods

```
void
  empty;
```

Undocumented.

```
void
  freeze;
```

Undocumented.

```
void
  add Comparable object;
```

Add an object.

Note that adding elements one-by-one to a `MutableSortedObjectArray` will work as an insertion sort, with quadratic performance. Use `addElementsFrom` with another sorted set instead, as it is much more efficient. Also, do not use `addElementsFromEnumerator` unless you know that the enumerator is sorted (and then use `addElementsFromSortedEnumerator`.

```
void
  addElementsFrom Collection object;
```

`addElementsFrom` performs in time proportional to the *sum* of the number of elements in both collections, if the other collection is a `SortedKeyed`. Otherwise the complexity is equal to the *product* of the number of the elements.

```
void
```

```
  addElementsFromSortedEnumerator Enumerator j
post
  [self verifySortedContents];
```

Undocumented.

```
void
  removeElementsFrom Collection object;
```

Undocumented.

```
void
  removeElementsFromSortedEnumerator Enumerator j
post
  [self verifySortedContents];
```

Undocumented.

```
void
  keepElementsFrom Collection object;
```

Undocumented.

File tom/State

class tom.State

inherits

Behaviour supers: `All`

variables

```
State isa;
```

Our class. The ‘State’ class is an instance of the State meta-class. The same is true for every other class.

```
private int asi;
```

Information used, in cunning ways, by the runtime.

methods

```
instance (id)
  alloc;
```

Return a newly created instance of the receiving class. All values, apart from the `isa` will have been initialized to their default value.

```
instance (id)
  alloc int size;
```

Like plain `alloc`, but allocate space for `size` bytes instead of the size of an instance. `size` is rounded up if it is not large enough for an instance of the receiving class to fit.

```
int
  instanceSize;
```

Return the size of the instances of this class. This may change due to dynamic loading if no instance has yet been created.

```
boolean
  classp;
```

Return `TRUE`, since we're a class object.

```
boolean
  coding-permanent-object-p;
```

Return `YES`. This should not be changed; this method is used in situations where it is not known whether the object is a class or not. In other situations, where it is known to be a class, this method is not invoked as it is known to return `YES`.

```
boolean
  dump_simple_p;
```

Return `YES`.

```
class (id)
  kind;
```

Return the class of the receiving object, i.e. the value of `isa`.

```
ByteString
  name;
```

Return the name of this class.

```
int (n)
  num_instances;
```

Return the number of currently live direct instances of this class, non-transitively.

```
Unit (unit)
  unit
post
  unit != nil;
```

Return the unit of this class. The postcondition states that such a unit must exit.

```
instance (id)
  new;
```

Return a newly created and initialized instance of the receiving class.

```
OutputStream
  write OutputStream s;
```

Write a description of this class object to the stream `s`.

instance tom.State

inherits

Behaviour supers: `All`

variables

```
class (id) isa;
```

Our class.

```
private int asi;
```

Information used, in cunning ways, by the runtime.

methods

```
boolean
  classp;
```

Return `FALSE`, as we're an instance, and not a class.

```
protected void
    dump MutableKeyed done
  indent MutableByteString prefix
  simple boolean allow_simple
   level int level
      to OutputStream s;
```

Hard worker for `dump`.

```
id
  init;
```

Designated initializer. Does nothing.

```
class (id)
  kind;
```

Return the class of the receiving object.

```
void
  set_kind class (State) a_class;
```

Change the class of the receiving object (i.e., the `isa`) into the `a_class`. Currently both the original and the new class must carry exactly the same state. Looser restrictions could be implemented...

```
OutputStream
  write OutputStream s;
```

Write the class and address of the receiving object to the stream `s`.

```
OutputStream
  writeFields OutputStream s;
```

Subsidiary for `write` to allow subclasses to write their fields to the stream `s`. The default implementation does nothing.

```
void
  dealloc;
```

Invoked by the garbage collector when an object has become garbage. Some important notes apply to this method:

Do not message any other objects from within this method as they might have become garbage too.

Since class objects can not become garbage, it is safe to message class objects.

When overriding this method, it is not necessary to invoke `State`'s implementation.

```
void
  gc_mark_elements;
```

This method is invoked by the garbage collector for instances which employ pointer typed instance variables, to have the receiving object mark the elements it references through said pointers. The default implementation marks the object referencing variables.

class tom.Recyclable

Recyclable is the class meant to help in cases where normal garbage collection provides suboptimal performance. It allows (but it does NOT require) manual object deallocation. To use, a simple call to [Recyclable recycle] frees the object. Note that there is no error checking, and calling methods from a recycled object results in an undefined behaviour.

Note that recycling is not mandatory. Unrecycled objects that are not needed any longer are freed in the normal GC passes. It is therefore not a good idea to spend too much effort (both development and runtime) on locating the objects that should be recycled - it is then more efficient to just rely on GC, and it is also less bug-prone.

Also note that not all the classes should be Recyclable - it is reserved for the exceptional cases, it may cause memory wastage, and in presence of generational GC, it may actually reduce performance.

inherits

State supers: `State`

variables

```
MutableObjectArray recycle_bin;
```

The `Array` with the recycled objects of this class.

methods

```
void
  load Indexed args;
```

Initialise the `recycle_bin`.

```
instance (id)
  alloc;
```

Override on [`State` alloc], this method tries to reuse a recycled object before allocating a fresh one.

instance tom.Recyclable

methods

```
void
  recycle;
```

Recycle the receiving object, allowing its reuse. Since this will not zero out the memory reserved for the object (or uninitialize it in any other way, it might be expedient (but not mandatory) to zero out the pointers from the object, so that GC can catch the referenced memory.

File tom/StreamBuffer

class tom.BufferedStream

inherits

State supers: `StreamStream`, `InputOutputStream`, `Conditions`

variables

```
const DEFAULT_BUFFER_SIZE = 8192;
```

The default value of the default size for the buffers of our instances.

```
static int default_buffer_size;
```

The default size for the buffers of our instances. If anyone sets this to a negative number, s/he should be ni'd.

methods

```
void
  load MutableArray arguments;
```

Undocumented.

instance tom.BufferedStream

variables

```
MutableByteArray buffer;
```

The buffer we use.

```
int num;
```

The number of elements in the `buffer`.

```
int next;
```

The index of the first character not yet handled (i.e. read or written).

methods

```
id
  init Stream s;
```

Initialize the newly allocated instance to buffer the stream `s` with a buffer sized the `default_buffer_size`.

```
id
        init Stream s
  bufferSize int cap;
```

Designated initializer. Initialize the newly allocated instance to buffer the stream `s` with a buffer sized `cap`.

```
int
  peek;
```

Return the value of the next byte to be returned by `read`, or -1 upon an error or end-of-file. This does not actually read the byte.

```
void
  unget byte b;
```

Stuff the byte `b` back (sort-of) into the stream. It will be the next byte to be read.

```
id
  flushOutput;
```

Flush any bytes buffered to the stream this instance is buffering.

```
void
  write byte b;
```

Write the byte `b`, raising a `stream-error` on error.

```
int
  write byte b;
```

Write the byte `b`, returning 1 upon success.

```
int
  writeBytes int length
        from pointer address;
```

Write to this stream the `length` bytes residing in memory at `address`.

```
byte
  read;
```

Return the next byte, raising a `stream-eos` upon an error or end-of-file.

```
int
  read;
```

Return the value of the next byte read, or -1 upon an error or end-of-file.

```
int (num_read)
  readRange (int, int) (start, length)
       into MutableByteArray destination;
```

Read at most `length` bytes into the `destination`, writing them from `start`. Return the number of bytes actually read.

```
protected int
  readBuffer;
```

Fill the buffer by reading more bytes from the `stream`. Return the number of bytes read.

```
protected int
  writeBuffer;
```

Write any bytes needing to be output to the `stream`. Return the number of bytes written.

File tom/String

class tom.String

inherits

State supers: `Indexed`, `Comparable`

instance tom.String

methods

```
deferred redeclare String
  frozen;
```

We'll return a `String` when frozen.

```
boolean
  dump_simple_p;
```

Return `YES`.

```
OutputStream
  dump_simple OutputStream s;
```

Print the receiving string, quoted.

```
deferred redeclare boolean
  equal String other;
```

Compare the receiving `String` with the other `String`.

```
deferred boolean
  equalByteString ByteString other;
```

Compare the receiving `String` with the other `ByteString`.

```
deferred boolean
  equalCharString CharString other;
```

Compare the receiving `String` with the other `CharString`.

```
deferred boolean
  equalUniqueString UniqueString other;
```

Compare the receiving `String` with the other `UniqueString`.

```
boolean
  equalModuloCase String other;
```

Compare the receiving String with the other String, ignoring case differences.

```
int
  compare id other;
```

Compare the receiving String with the other.

```
(int, int)
  rangeOfString String string
         range: (int, int) (start, len) = (0, -1);
```

Return the range of the occurrence of the string in the receiving string. Return a negative length in case it could not be found. The optional start and length can be specified to restrict the searching within the receiving string.

```
MutableArray
  componentsSeparatedBy char c
                 limit: int limit = -1
          excludeEmpty: boolean excl = NO
     substringSelector: selector sel = selector (String substring (int, int));
```

Return a (mutable) Array of strings, taken from the receiving string by splitting it at characters with the indicated char value. Thus, splitting '/usr/tmp' at each '/' returns an array holding the empty string, 'usr', and 'tmp'.

The optional argument limit specifies the maximum number of items in which the caller is interested, or -1 for all items. For example, if '/usr/foo/bar' is split on '/' in 3 items, the array returned contains '', 'usr', and 'foo/bar'.

The optional argument excl, if YES specifies that zero-length substrings are not to be included in the result. Thus, splitting '/aap/noot/mies/wim' in 3 items, ignoring empty items, returns an array containing 'aap', 'noot', and 'mies/wim'.

The optional selector sel specifies the method to be called to extract the substrings from the receiving string. The default selector is "r_substring_(ii)". To retrieve mutable substrings, the selector "r_mutableSubstring_(ii)" could be used.

```
deferred MutableString
  mutableSubstring (int, int) (start, len)
pre
  start >= 0 && len >= -1;
```

Return a MutableString holding the characters from the receiving String in the (clipped) range (start, len).

```
deferred String
  substring (int, int) (start, len)
pre
  start >= 0 && len >= -1;
```

Return a constant String holding the characters from the receiving String in the (clipped) range (start, len).

```
deferred UniqueString
  uniqueString;
```

Return a unique version of the receiving string. Do not use this method to create unique strings; use [UniqueString with my_string] instead. (This method only creates strings which think they are unique; the UniqueString class ensures they actually are.)

```
id
  downcase;
```

Undocumented.

```
id
  upcase;
```

Undocumented.

```
double (value)
  doubleValue;
```

Return the double value at the start of the string.

```
(int, boolean, int) (value, full_range, actual_length)
    integerValue (int, int) (start, len)
    defaultBase: int base = 10
      allowSign: boolean signs = YES
    allowCBases: boolean c_bases = YES
  baseSeparator: byte base_separator = '_'
    decimalBase: boolean decimal_base = YES;
```

Convert the number contained in the receiving string from index start, running for len bytes (which -1 for unlimited length).

The value returned is a tuple (extracted value, occupied full range, actual length). If the actual length is 0, the extracted number will be 0.

The base defaults to 10, but can be any number. If it is larger than 10, alpha characters encountered have the value of 11 + the offset from the alpha character to the start of its range. Thus, ‘a’ is 10, ‘z’ is 35.

Iff signs, a leading ‘+’ or ‘-’ sign is accepted.

Iff c_bases, C-style base indicators may be used: a number starting with a ‘0’ denotes an octal number; a number starting with ‘0x’ or ‘0X’ is a hexadecimal number.

Iff the base_separator is not 0, a number can be prefixed with a base indication followed by the base separator to specify the base of the actual number to follow. The base is read using the a decimal base, unless decimal_base is FALSE, in which case the base is read in the default base. Thus, '10_10', with '_' as a base separator, returns `base` if `decimal_base` is `FALSE`, and 10 if it was `TRUE`.

```
int
  intValue;
```

Simple front-end for `integerValue` (with default arguments).

```
int
  unsignedIntValue;
```

Simple front-end for `integerValue`, similar to `intValue`, but not allowing a negative value. For a negative value entered (due to `integerValue` not doing overflow checking), 0 is returned.

```
boolean
  isAlpha char c;
```

Return TRUE iff the character `c` denotes a letter.

```
boolean
  isDigit char c;
```

Return `TRUE` iff the character `c` is a digit.

```
boolean
  isLower char c;
```

Return `TRUE` iff the character `c` is in lower-case.

```
boolean
  isPunct char c;
```

Return `TRUE` iff the character `c` is a punctuation character.

```
boolean
  isSpace char c;
```

Return `TRUE` iff the character `c` is a space character.

```
boolean
  isUpper char c;
```

Return `TRUE` iff the character `c` is in upper-case.

```
char
  toLower char c;
```

Undocumented.

```
char
  toTitle char c;
```

Undocumented.

```
char
  toUpper char c;
```

Undocumented.

```
int
  digitValue char c;
```

Undocumented.

```
int
  alphaValue char c;
```

Undocumented.

```
id
  stringByDecoding String encoding_name;
```

Return a string by decoding it assuming it was encoded using the encoding named by `encoding_name`. The default implementation simply returns `self`.

File tom/StringStream

class tom.StringStream

A `StringStream` is sort-of an enumerator on a `String`, with a `InputStream` interface.

inherits

State supers: `State`, `InputStream`

methods

```
instance (id)
  with String string;
```

Return a new stream on the `string`.

instance tom.StringStream

variables

```
String string;
```

The `String` we're streaming.

```
int next;
```

The index of the next byte to read.

methods

```
protected id
  init String s;
```

Designated initializer.

```
byte
  read;
```

Undocumented.

```
int
  read;
```

Undocumented.

```
int
  readRange (int, int) (start, num)
       into MutableByteArray buffer;
```

Read the range `(start, num)` from the string into the `buffer`.

File tom/Thread

class tom.Thread

The `Thread` class provides an abstraction to the multi-threading facilities provided by the underlying operating system.

A new thread is started by the `performInThread with` method provided by the instance `All`. The value returned by that method is the `Thread` object of the newly created thread.

Every thread has an id. The id of the current thread is available from the `Thread` class as `current_id`. The main thread (which every program has, even when running single-threaded) has id 0. Due to the differences in target implementations, exiting the main thread, by invoking that `Thread`'s `exit` method, is not guaranteed not to exit the program.

Multi-threading need not be available on all TOM targets. The `functioning` method returns `FALSE` on those targets on which multi-threading is not available. However, a TOM programmer can assume multi-threading to always be available.

inherits

State supers: `State`

variables

```
static MutableEqSet threads;
```

The currently existing threads.

```
local static public instance (id) current;
```

The current thread.

```
local static public int current_id;
```

The id of the current thread.

methods

```
boolean
  functioning;
```

Return `TRUE` iff we can run multiple threads on this target.

```
Set
  threads;
```

Return the currently existing threads.

instance tom.Thread

variables

```
public int thread_id;
```

The TOM thread id of this thread.

methods

```
void
  exit int rc
pre
  self == [Thread current];
```

Exit the receiving thread, which must be the current thread.

```
protected id (self)
  init int th_id;
```

Designated initializer.

File tom/Trie

class tom.Trie

The `Trie` is a class providing the mechanism to store information in a trie on `char` strings. It does not by itself store any information, subclasses should be created to hold the information. The `Trie` is accompanied by its subclass `ObjectTrie` which can store objects in a trie.

inherits

State supers: `State`, `Constants`

instance tom.Trie

variables

```
int start;
```

The offset to the first element in `next`, i.e. the element with numeric value `start` resides at index 0 in `next`.

```
int beyond;
```

The value of the first element beyond the last element in `next`.

```
Any next;
```

If `start == beyond`, this is the suffix `String` which leads up to the value this node holds (if any). Otherwise, `start > beyond` and this is a `MutableObjectArray` pointing to the next nodes, and which is to be indexed with offset `start`.

methods

```
deferred boolean
  isEmpty;
```

Return `YES` iff we can hold a value, i.e. if we do not yet hold a value.

```
protected id
  createNode String str
       start int s
         end int e
     options int options;
```

Create the node for that part of the string `str` starting at `s`, and ending at `e`. The `TRIE_LOOKUP_PREFIX` option is ignored. If this node already exists, it is returned.

When the `options` include `TRIE_FOLD_CASE`, the `str` is inserted in lower case.

```
protected id
  findNode String str
```

```
     start int s
       end int e
   options int options;
```

Find the node for that part of the string `str` starting at `s`, and ending at `e`. Iff a prefix match is desired, the node returned is the longest prefix match.

```
protected void
  pushSuffix int options;
```

Push our suffix one node down.

```
OutputStream
  write OutputStream s;
```

Undocumented.

```
deferred OutputStream
  writeValue OutputStream s;
```

Undocumented.

class tom.ObjectTrie

An `ObjectTrie` is a `Trie` which can hold an object.

inherits

State supers: `Trie`

instance tom.ObjectTrie

variables

```
public mutable Any value;
```

Our value.

methods

```
boolean
  isEmpty;
```

Undocumented.

```
void
  pushSuffix int options;
```

Move our value with the suffix.

```
Any
```

```
       at String key
  options int options;
```

Undocumented.

```
void
      set All object
       at String key
  options int options;
```

Undocumented.

```
OutputStream
  writeValue OutputStream s;
```

Undocumented.

```
Any
  at String key;
```

Undocumented.

```
void
  set All object
   at String key;
```

Undocumented.

File tom/TypeDescription

class tom.TypeDescription

inherits

State supers: `State`

variables

```
static MutablePointerDictionary descriptions;
```

A container holding the mapping from `struct trtd_selector_args` to a `TypeDescription` instance.

methods

```
instance (id) (result)
  for pointer args;
```

Designated allocator, using a cache.

instance tom.TypeDescription

variables

```
public pointer types_description;
```

The internal runtime structure to the type description.

methods

```
boolean
  equal id other;
```

Undocumented.

```
protected id (self)
  init pointer args;
```

Designated initializer.

```
int
  length;
```

Return the number of elements.

```
int
  component int n;
```

Describe the element at `n`, indexed 0. This returns one of the `TYPEDESC_* Constants`.

```
OutputStream
  writeFields OutputStream s;
```

Describe the component types.

File tom/Unicoding

class tom.Unicoding

The `Unicoding` class object maintains information on the `Unicode` character coding.

inherits

Behaviour supers: `All`

variables

```
static ByteArray is_digit;
```

Bitmap for digit predicate.

```
static ByteArray is_letter;
```

Bitmap for letter predicate.

```
static ByteArray is_lower;
```

Bitmap for lower predicate.

```
static ByteArray is_punct;
```

Bitmap for punctuation predicate.

```
static ByteArray is_space;
```

Bitmap for space predicate.

```
static ByteArray is_upper;
```

Bitmap for upper predicate.

methods

```
protected ByteArray
  loadPredicateSet String predicate
       alternative selector alt_sel;
```

Load and return the predicate set for the `predicate` on Unicode characters. If it can not be located, the `alt_sel` is used to extract part of the information needed from the `USASCIIEncoding`.

```
boolean
  isAlpha char c;
```

Undocumented.

```
boolean
  isDigit char c;
```

Return `TRUE` iff the character `c` is a digit according to the encoding of the receiving string.

```
boolean
  isLower char c;
```

Undocumented.

```
boolean
  isPunct char c;
```

Undocumented.

```
boolean
  isSpace char c;
```

Undocumented.

```
boolean
  isUpper char c;
```

Undocumented.

```
char
  toLower char c;
```

Undocumented.

```
char
  toTitle char c;
```

Undocumented.

```
char
  toUpper char c;
```

Undocumented.

```
int
  digitValue char c;
```

Undocumented.

```
int
  alphaValue char c;
```

Undocumented.

instance tom.Unicoding

inherits

Behaviour supers: `All`

File tom/Unit

class tom.Unit

inherits

State supers: `State`

variables

```
static MutableDictionary units;
```

All units known.

methods

```
instance (id)
  named String name;
```

Return the `Unit` with the `name`, or `nil` if said unit does not currently exist.

```
protected void
  fillUnits
pre
  !units;
```

Create the `units` dictionary and fill it with the currently known units.

```
Mapped
  units;
```

Return the collection of units, keyed on their name.

instance tom.Unit

variables

```
public String name;
```

The name of this unit.

```
Dictionary classes;
```

The classes in this unit, keyed on their unqualified name.

methods

```
protected id (self)
  initWithName String n
       classes Dictionary c;
```

Designated initializer.

```
class (State)
  classNamed String name;
```

Return the class with the given unqualified `name`, or `nil` if a class with that name does not exist in this unit.

File tom/XL

class tom.XLTokens

The tokens available from XL.

variables

```
const XLT_PAR_CLOSE = -9;
```

```
const XLT_PAR_OPEN = -8;
```

```
const XLT_DOUBLE = -7;
```

```
const XLT_FLOAT = -6;
```

```
const XLT_LONG = -5;
```

```
const XLT_INT = -4;
```

```
const XLT_SYMBOL = -3;
```

```
const XLT_EPSILON = -2;
```

```
const XLT_EOF = -1;
```

instance tom.XLTokens

class tom.XL

inherits

State supers: State, XLTokens

variables

`const XLS_SYMBOL = 0;`

Different states of the lexer state machine. Basically, these states are the states of reading a floating point number, with a prefix for an integer, and an escape for a non-numeric input.

`const XLS_SIGN = 1;`

`const XLS_INT = 2;`

`const XLS_DOT = 3;`

`const XLS_FRAC = 4;`

`const XLS_EXP_E = 5;`

`const XLS_EXP_SIGN = 6;`

`const XLS_EXP = 7;`

instance tom.XL

variables

`public InputStream stream;`

The stream being lexed.

`MutableByteString buffer;`

The buffer used for building the text of the token.

`public long int_value;`

The most recent integer value retrieved.

`public double float_value;`

The most recent floating value retrieved.

```
public int current_line;
```

The current line.

```
public int token;
```

The current token.

```
int next_char;
```

The next character, i.e. the first character of the next token. This is `XLT_EOF` for end of stream, or `XLT_EPSILON` if this should be considered invalid (and read before starting the next token).

methods

```
id
  initWithStream InputStream s;
```

Designated initializer.

```
int
  intValue;
```

Return the `int_value` as an int. Any loss of bits is not remarked.

```
MutableString
  matched;
```

Return the matched text.

```
int
  nextToken;
```

Skip space and return the next token.

```
(pointer, int) (contents, length)
  readBytes int expected_length
post
  length == expected_length;
```

Skip whitespace, read a quoted string of bytes (`"quoting \\like\\ \"this\""`) and return it. The length should match the expected length. Anything unexpected results in the return of a `NULL` pointer.

File tom/archiving

class tom.StreamEncoder

inherits

State supers: `Encoder`

instance tom.StreamEncoder

variables

```
OutputStream stream;
```

The stream to which we write.

methods

```
id
  initWithStream OutputStream s;
```

Undocumented.

```
protected State
  replacementObjectFor State object;
```

Undocumented.

class tom.StreamDecoder

inherits

State supers: `Decoder`

instance tom.StreamDecoder

variables

```
InputStream stream;
```

The stream from which we read.

methods

```
id
  initWithStream InputStream s;
```

Designated initializer.

class tom.BinaryStreamEncoder

inherits

State supers: `StreamEncoder`, `BinaryEncoder`

instance tom.BinaryStreamEncoder

methods

```
id
  initWithStream OutputStream s;
```

Designated initializer.

```
void
  finishEncodingRoot All object;
```

Finish the graph.

```
protected void
  writeByte byte b;
```

Undocumented.

```
protected void
  writeBytes (int, int) (start, length)
        from ByteArray r;
```

Undocumented.

```
protected void
  writeBytes (pointer, int) (address, length);
```

Undocumented.

class tom.BinaryStreamDecoder

inherits

State supers: `StreamDecoder`, `BinaryDecoder`, `C`

instance tom.BinaryStreamDecoder

methods

```
id
  initWithStream InputStream s;
```

Designated initializer.

```
protected byte
  readByte;
```

Undocumented.

```
protected void
```

```
  readBytes int num
         to pointer address;
```

Undocumented.

class tom.TextStreamEncoder

inherits

State supers: `StreamEncoder`

instance tom.TextStreamEncoder

methods

```
void
  startEncodingRoot All object;
```

Output the top of the graph.

```
void
  finishEncodingRoot All object;
```

Finish the graph.

```
class (State)
  startEncoding State object;
```

Output the start of the `object`.

```
void
  finishEncoding State object;
```

Finish the output of the `object`.

```
protected int
  identityForClass class (State) a_class;
```

Identify this class on the output `stream`, reporting its coding version.

```
void
  encodeNilObject;
```

Output `()`, which is the notation for the `nil` object.

```
void
  encodeReference int v;
```

Undocumented.

```
void
  encode boolean v;
```

Undocumented.

```
void
  encode byte v;
```

Undocumented.

```
void
  encode char v;
```

Undocumented.

```
void
  encode int v;
```

Undocumented.

```
void
  encode long v;
```

Undocumented.

```
void
  encode float v;
```

Undocumented.

```
void
  encode double v;
```

Undocumented.

```
void
  encode selector v;
```

Undocumented.

```
void
  encodeBytes (int, int) (start, length)
         from ByteArray r;
```

Undocumented.

class tom.TextStreamDecoder

This class is unimplemented.

inherits

State supers: `StreamDecoder`, `XLTokens`

instance tom.TextStreamDecoder

variables

`XL lexer;`

The lexer actually doing the reading from our `stream`.

`int token;`

The current token, cached so we know when we are starting up, in which case the token is `XLT_EPSILON`.

methods

```
id
  initWithStream InputStream s;
```

Designated initializer of our super.

```
id
  initWithLexer XL l;
```

Designated initializer.

```
Any
  decode;
```

Undocumented.

```
byte
  decode;
```

Undocumented.

```
boolean
  decode;
```

Undocumented.

```
char
  decode;
```

Undocumented.

```
int
  decode;
```

Undocumented.

```
long
  decode;
```

Undocumented.

```
float
  decode;
```

Undocumented.

```
double
  decode;
```

Undocumented.

```
(pointer, int) (contents, length)
  decodeBytes;
```

Undocumented.

```
protected void
  declareClass;
```

Undocumented.

```
protected int
  nextToken;
```

Undocumented.

```
protected Any
  readReference;
```

Undocumented.

```
protected void
  skipList;
```

Read tokens up to and including the first top-level close parenthesis.

```
protected void
  termSymbol String name;
```

Undocumented.

```
protected void
  termToken int t;
```

Undocumented.

File tom/behaviours

class tom.Comparable

inherits

Behaviour supers: All

instance tom.Comparable

inherits

Behaviour supers: All

methods

```
deferred int
  compare id other
pre
  other != nil;
```

Return 0 if the other is considered equal by the receiving object. 1 if the receiver considers himself larger, and -1 when smaller.

```
void
  set_index int index
    in_heap Heap h;
```

Functionality used by Heap to keep track of the index of its elements. Instances of HeapElement actually remember the index. The default implementation just ignores it.

```
int
  index_in_heap Heap heap;
```

Return the index of this element in the heap. Instances of HeapElement can do this O(1) instead of the O(n) of Comparable. On the other hand, Comparable can reside in any number of Heap, and it is only removal other than through root extraction which has become slower.

class tom.Container

A container is an object which gets to mark its elements after normal marking has been done. This is very usable for unique string tables, DO proxies, etc; actually: all cases where an object having become garbage implying it should be removed from its container, and the container itself is not allowed to reference the object in a normal way (since then it would never become garbage).

In short, a the combo of garbage collector and container implements weak referencing.

instance tom.Container

methods

```
deferred void
  gc_container_mark_elements;
```

The container mark method.

```
boolean
  isContainer;
```

Return `TRUE` iff the receiving object is a container.

```
void
  setIsContainer boolean container_p;
```

Set this object to be a container, or not, depending on `container_p`.

```
void
  setStackNotify boolean notify_p;
```

State that this container wants to be notified when it is conservatively pinpointed.

```
boolean
  wantsStackNotify;
```

Does this container receive stack notifications?

```
void
  gc_stack_notify;
```

Be notified of a reference from the stack, as request by `setStackNotify`. Default implementation does nothing.

class tom.Copying

The `Copying` class defines an interface to copying objects.

`Copying` inherits from `State` since class objects should not be copyable. Inheriting from `State` ensures that the `Copying` instance methods can not be inherited by class objects.

inherits

State supers: `State`

instance tom.Copying

methods

```
id
  copy;
```

Return a shallow copy of the receiving object.

```
id
  deepCopy;
```

Return a deep copy of the receiving object.

```
id
    deepen int level
  mutably: boolean mutable_p = NO;
```

Intended to be called on a recently acquired copy of an object, `deepen mutable:` deepens the copy. Iff the optional `mutable_p` is `TRUE`, the deepened copies will also be mutable. The default implementation does nothing.

The `level` should be less than 0 for an infinite deepen. `length == 0` is a nop; iff `length > 0`, every element of the copy is copied and deepened with `level - 1`.

The value returned is `self`.

```
id
  initCopy;
```

Initialize the receiver just after it has been created as the result of a `copy`. The default implementation does nothing but return `self`.

```
id
  initAsCopyOf All other;
```

Initialize the receiver just after it has been created as the result of a `mutableCopy` of the `other` object. The default implementation does nothing but returning `self`.

```
Any
  mutableCopy;
```

Return a mutable (shallow) copy of the receiving object. For objects which do not discern between mutable and immutable variants, the default implementation returns `[self copy]`.

Mutable copying asks the receiving object for its `mutableCopyClass`. If this class is `isa`, `self` is sent a `copy`. Otherwise, an instance of the class is allocated and sent an `initAsCopyOf`.

```
class (State)
  mutableCopyClass;
```

Return the class of the object resulting from a mutable copy of this object. The default implementation simply returns `isa`.

class tom.Enumerable

inherits

State supers: `State`

methods

```
instance (id)
  withEnumerable Enumerable other;
```

Invoke `self`'s `withEnumerator` with an `enumerator` from the `other`.

```
instance (id)
  withEnumerator Enumerator e;
```

Return a newly allocated instance of the receiving class, filled with the elements from the `Enumerator` `e`.

instance tom.Enumerable

methods

```
deferred protected id
  initWithEnumerator Enumerator e;
```

Initialize with the elements from the `Enumerator` `e`.

```
deferred Enumerator
  enumerator;
```

Return an `Enumerator` on the receiving object.

class tom.Enumerator

instance tom.Enumerator

methods

```
deferred (boolean, Any)
  next;
```

Return a tuple containing the next object, preceded by a boolean value indicating whether the end of the enumerable has been reached; if the boolean is `TRUE`, the end has not yet been reached.

```
(boolean, byte)
  next;
```

Default implementations for direct value retrieving enumerators.

```
(boolean, char)
  next;
```

Undocumented.

```
(boolean, int)
  next;
```

Undocumented.

```
(boolean, long)
  next;
```

Undocumented.

```
(boolean, float)
  next;
```

Undocumented.

```
(boolean, double)
  next;
```

Undocumented.

```
(boolean, pointer)
  next;
```

Undocumented.

class tom.MapEnumerator

inherits

State supers: `Enumerator`

instance tom.MapEnumerator

methods

```
deferred (boolean, Any, Any)
  next;
```

MapEnumerator allows iteration over both keys and values.

File tom/coding

class tom.ObjectCoder

ObjectCoder is a workaround for circular hierarchy - it is only a placeholder for methods called from `State (Coding)`.

inherits

State supers: `Conditions`, `Constants`

methods

```
void
  encodeObject State obj
    usingCoder Encoder coder;
```

Undocumented.

```
void
  initObject State obj
  usingCoder Decoder coder;
```

Undocumented.

instance tom.ObjectCoder

class tom.State (Coding)

This extension of `State` defines the functionality for encoding and decoding objects. To be able to encode an object, it must at least properly implement `encodeUsingCoder`. Similarly, to be decodable, it must implement `initWithCoder`.

The unit of archiving is a class, not an extension. This means that if an extension adds state information which needs to be archived (or encoded onto a `too.PortCoder`), the extension must re-implement the coding methods.

variables

```
public mutable boolean encode_simply;
```

Classes that want to be encoded in the obvious way, by writing the values of their variables, set this to TRUE.

methods

```
int
  version;
```

Return the current version of the class `cls`. This is the version that will be written when coding instances of this class or a subclass thereof. The default version is 0.

A version should only be returned if `self` is identical to the class containing the method definition, i.e. the method is not inherited. Otherwise, the two are unequal, and the version of a subclass is requested that does not implement this method, and hence should return version 0.

```
boolean
  never-encode-simply-p;
```

Return `YES` if `encode_simply` of all classes involved in the receiving object will always return `FALSE`. Coding is sped up tremendously in that case. The defaultis `NO`, to not speed up and allow for passive encoding.

```
boolean
  persistent-coding-p;
```

Return `YES`.

instance tom.State (Coding)

methods

```
class (State)
  classForCoder Encoder coder;
```

Return the class to be put in the coded stream as the class of this object. The default implementation simply returns `isa`, which is the receiving object's class.

```
void
  encodeUsingCoder Encoder coder;
```

Encode the receiving object to the target `coder`. Every object should first invoke this method of all its direct superclasses before encoding its instance variables, but only if `hasBeenCodedFor` for the class implementing the method returns `FALSE`. For classes that set `encode_simply` to TRUE, this method will use introspection to encode the class variables.

```
boolean
  never-encode-simply-p;
```

Return `YES` if `encode_simply` of all classes involved in the receiving object will always return `FALSE`. Coding is sped up tremendously in that case.

```
boolean
  persistent-coding-p;
```

Return `NO`.

```
Any (self)
  replacementForStreamCoder StreamEncoder coder;
```

Return the object to be encoded on the `StreamEncoder` `coder` (i.e. archived or wired) instead of the receiving object. The default implementation simply returns `self`.

```
void
  initWithCoder Decoder coder;
```

Initialize the receiving object from the `coder`. After verifying that this method implementation has not yet been invoked (using `hasBeenCodedFor`), this method should invoke the implementation of this method by the superclasses, followed the fields that were encoded by this class. Decoding must be done in the same order as encoding. Default implementation on `State` will use introspection to init the objects that requested it by setting `encode_simply` on their class to TRUE.

Note that this method returns `void`. An object can change the actual object returned from decoding by implementing `awakeAfterUsingCoder`.

```
id (self)
  awakeAfterUsingCoder Decoder coder;
```

Return the object to be the object retrieved from decoding instead of the receiving object. The default implementation returns `self`.

Objects can use this method to return their administered counterpart, like `UniqueString` objects do.

Note that if an object is referenced during its decoding (i.e. object A is referenced by an object B which is decoded because B is (indirectly) referenced by A), it must not return a different object from `awakeAfterUsingCoder`. If it does, a `coding-condition` is raised.

class tom.Coder

inherits

State supers: `State`, `Conditions`

methods

```
int
  version;
```

The version of the coding scheme used. The current version is 0.

instance tom.Coder

methods

```
void
  willCodeVariable String name
         forObject All object
       inExtension Extension x;
```

Notify the coder object about the variable being coded. This allows primitive form of state versioning control. Note that there are no guarantees that every class will send this notification while encoding itself.

```
void
  willCodeExtension Extension x
          forObject All object;
```

Notify the coder of the extension being coded. There are no guarantees that every class will send this notification.

```
void
  doneCodingExtension Extension x;
```

Notify the coder that the coding of the extension x is finished.

class tom.BinaryCoder

The `BinaryCoder` classes `BinaryEncoder` and `BinaryDecoder` can archive dearchive a graph of objects in a binary form onto/from a stream. The format is rather simple: Every item stored is preceded by a tag byte indicating what the next item is. There are a few secondary tags to introduce classes, etc.

Every instance or class written is internally numbered in the order the objects are written. References to these objects are encoded in the number of bytes necessary for the number of currently known objects. The secondary tags `2` and `4` switch to 2 and 4 byte reference encoding, respectively.

The `nil` object is denoted by the `0` tag.

Selectors are encoded as a tag `S`, followed by the assigned selector number (which is an `int`, starting at 1) and the corresponding `Selector` object. Selectors already encoded are denoted by a tag `s` and the `int` selector number. The invalid selector (the default value of `selector` typed variables, also available as `[Runtime nullSelector]`) is identified by the tag `s` followed by 0 as the selector number.

inherits

State supers: `Coder`

instance tom.BinaryCoder

variables

```
int reference_size;
```

The number of bytes issued for a reference. This starts with 1 (a byte), and can become 2 (a char) or 4 (an int).

methods

```
id
  init;
```

Undocumented.

class tom.Encoder

inherits

State supers: `Coder`

instance tom.Encoder

variables

`MutableEqDictionary tmp_objects_done;`

Keyed on the objects already encoded, the value is the identifier (which is an `IntNumber`) used for this object. This dict only contains temporary objects, i.e. objects that can be forgotten about after each `encodeRoot`.

`MutableEqDictionary perm_objects_done;`

Similar, the non-temporary objects. This includes class objects and `Selector` objects.

`MutableEqSet objects_skipped;`

The set of conditional objects that were skipped.

`MutableEqSet coded_classes;`

The classes which, for the current object, have already done their part in the coding.

`int last_object_id;`

The most recently issued object identifier.

methods

```
void
  encodeRoot State object;
```

The main entry `Encoder` method: encode the `object` and the whole object graph implied by it. This method is not reentrant.

```
id
  init;
```

Designated initializer.

```
boolean
  hasBeenCodedFor class (State) the_class;
```

Return `NO` if the object currently being encoded on this coder has not yet been encoded for `the_class`. Return `YES` otherwise. While coding an object, only the first invocation for a certain `the_class` will return `YES`; subsequent invocations will return `NO`.

```
void
  encode State object;
```

Encode the `object`, unconditionally.

```
void
  encodeConditionally State object;
```

Encode the `object`, but only if it already is part of the output graph. If this is not the case, `nil` is encoded, and if later on in the coding process the object previously encoded as nil is encountered (unconditionally), a `program-condition` will be raised to flag the inconsistency.

```
deferred void
  encode boolean v;
```

Encode the `boolean` `v`.

```
deferred void
  encode byte v;
```

Encode the `byte` `v`.

```
deferred void
  encode char v;
```

Encode the `char` `v`.

```
deferred void
  encode int v;
```

Encode the `int` `v`.

```
deferred void
  encode long v;
```

Encode the `long` `v`.

```
deferred void
  encode float v;
```

Encode the `float` `v`.

```
deferred void
  encode double v;
```

Encode the `double` `v`.

```
deferred void
  encode selector v;
```

Encode the `selector` `v`.

```
deferred void
  encodeBytes (int, int) (start, length)
         from ByteArray r
pre
  start >= 0 && length >= -1;
```

Encode the bytes in the range (start, length) from the array r.

```
deferred void
  encodeBytes (pointer, int) (address, length);
```

Encode the length bytes of which the first one resides at the address.

```
deferred protected State
  replacementObjectFor State object;
```

Return the object to be encoded to this coder instead of the object. This method is implemented by subclasses to retrieve the actual object from the object itself, for instance by asking for it replacementForStreamCoder or replacementForPortCoder.

```
deferred protected void
  encodeNilObject;
```

Encode the nil reference.

```
deferred protected void
  encodeReference int v;
```

Encode a reference to the object known as v.

```
protected int
  identityFor All object;
```

Return the identity to be used for the non-class object. This returns the next value of last_object_id.

```
protected int
  identityForClass class (State) a_class;
```

Return the identity to be used for the class object a_class. This returns the next value of last_object_id.

```
class (State)
  startEncoding State object;
```

Undocumented.

```
protected void
  finishEncoding All object;
```

Invoked when the object has been encoded. Default does nothing.

```
protected void
  startEncodingRoot All object;
```

Invoked when coding starts with the root object. Default does nothing.

```
protected void
  finishEncodingRoot All object;
```

Invoked when coding the root `object` has finished. Default does nothing.

class tom.BinaryEncoder

inherits

State supers: `BinaryCoder`, `Encoder`

instance tom.BinaryEncoder

variables

```
MutableDictionary selectors;
```

The selector dictionary, from `Selector` to `IntNumber`.

methods

```
id
  init;
```

Designated initializer.

```
class (State)
  startEncoding State object;
```

Undocumented.

```
protected void
  finishEncoding All object;
```

Invoked when the `object` has been encoded. Emit a close paren.

```
protected void
  updateReferenceSize;
```

Undocumented.

```
protected int
  identityFor All object;
```

Undocumented.

```
protected int
  identityForClass class (State) a_class;
```

Identify this class on the output `stream`, reporting its coding version.

```
protected void
  encodeNilObject;
```

Undocumented.

```
protected void
  encodeReference int v;
```

Undocumented.

```
void
  encode boolean v;
```

Undocumented.

```
void
  encode byte v;
```

Undocumented.

```
void
  encode char v;
```

Undocumented.

```
void
  encode int v;
```

Undocumented.

```
void
  encode long v;
```

Undocumented.

```
void
  encode float v;
```

Undocumented.

```
void
  encode double v;
```

Undocumented.

```
void
  encode selector v;
```

Undocumented.

```
void
  encodeBytes (pointer, int) (address, length);
```

Undocumented.

```
void
  encodeBytes (int, int) (start, length)
         from ByteArray r;
```

Undocumented.

```
protected void
  writeReference int r
pre
  reference_size == 1 || reference_size == 2 || reference_size == 4;
```

Undocumented.

```
deferred protected void
  writeByte byte b;
```

Undocumented.

```
deferred protected void
  writeBytes (int, int) (start, length)
        from ByteArray r;
```

Undocumented.

```
deferred protected void
  writeBytes (pointer, int) (address, length);
```

Undocumented.

```
protected void
  writeChar char c;
```

Undocumented.

```
protected void
  writeInt int i;
```

Undocumented.

```
protected void
  writeLong long l;
```

Undocumented.

class tom.Decoder

The `Decoder` class defines the interface to all decoder classes, be it binary or textual, stream or port base.

inherits

State supers: `Coder`

instance tom.Decoder

variables

```
MutableIntDictionary tmp_objects_done;
```

Objects, indexed on their identity, as retrieved from this coder.

```
MutableIntDictionary perm_objects_done;
```

```
IntegerRangeSet objects_referenced;
```

The identity of the objects that have been referenced while being decoded.

```
MutableEqDictionary class_versions;
```

Mapping from a class to the decoding version of that class.

```
MutableEqSet coded_classes;
```

The classes which, for the current object, have already done their part in the coding.

methods

```
id
  init;
```

Designated initializer.

```
Any
  decodeRoot;
```

This is the entry point for the user of this decoder. The user invokes `decodeRoot` to retrieve an object, plus its underlying graph, from this decoder. The object is returned.

```
boolean
  hasBeenCodedFor class (State) the_class;
```

Return `NO` if the object currently being decoded on this coder has not yet been decoded for `the_class`. Return `YES` otherwise. While coding an object, only the first invocation for a certain `the_class` will return `YES`; subsequent invocations will return `NO`.

```
int
  versionOfClass class (State) cls
pre
  class_versions[cls] != nil;
```

Return the version of the class `cls` as encountered by this coder. The version can only be retrieved of classes already encountered curing the decoding process.

```
deferred Any
  decode;
```

Retrieve an object from this decoder and return it.

```
deferred boolean
  decode;
```

Undocumented.

```
deferred byte
  decode;
```

Undocumented.

```
deferred char
  decode;
```

Undocumented.

```
deferred int
  decode;
```

Undocumented.

```
deferred long
  decode;
```

Undocumented.

```
deferred float
  decode;
```

Undocumented.

```
deferred double
  decode;
```

Undocumented.

```
deferred selector
  decode;
```

Undocumented.

```
deferred (pointer, int)
  decodeBytes;
```

Decode a sequence of bytes from the coder to newly allocated memory space. Return the address and the length.

```
deferred void
  decodeBytes int num
           to pointer address;
```

Decode the `num` bytes from the coder to the `address`.

```
protected Any
  decodeObject class (State) cls
            as int ref;
```

Undocumented.

```
protected void
  finishDecoding All o;
```

Invoked by `decodeObject as`, after having invoked `initWithCoder`, but before `awakeAfterUsingCoder`. The default implementation does nothing.

```
protected Any
  reference int i;
```

Return the object referenced as the number `i`.

class tom.BinaryDecoder

The `BinaryDecoder` is an abstract decoding class which can decode binary encoded objects. It serves as the decoding engine for the `BinaryStreamDecoder` and `too.PortDecoder`.

inherits

State supers: `BinaryCoder`, `Decoder`, `C`

instance tom.BinaryDecoder

variables

```
MutableArray selectors;
```

The selectors encountered so far, indexed on their identity.

methods

```
id
  init;
```

Designated initializer.

```
Any
```

```
  decode;
```

Decode and return an object.

```
Any
  decode byte b;
```

Decode and return an object, announced by the tag `b`.

```
boolean
  decode;
```

Undocumented.

```
byte
  decode;
```

Undocumented.

```
char
  decode;
```

Undocumented.

```
int
  decode;
```

Undocumented.

```
long
  decode;
```

Undocumented.

```
float
  decode;
```

Undocumented.

```
double
  decode;
```

Undocumented.

```
selector (result)
  decode;
```

Undocumented.

```
(pointer, int)
  decodeBytes;
```

Undocumented.

```
void
  decodeBytes int length
           to pointer address;
```

Undocumented.

```
protected void
  finishDecoding All o;
```

Invoked when the object `o` has been decoded. Read a close paren.

```
protected byte
  nextPrimary;
```

Return the next primary tag byte, handling secondary tags such as reference size changes and class declarations.

If an unknown class is encountered, a `unknown-class-condition` is signaled. A handler may return a replacement class to be used instead. Failure to do so will later on result in a `nil-receiver` condition or a failed precondition.

```
protected byte
  nextPrimary byte expected;
```

Return the next primary tag byte, which must match `expected`. If it doesn't, a `program-condition` is raised.

```
protected int
  readReference
pre
  reference_size == 1 || reference_size == 2 || reference_size == 4;
```

Read an object reference from this decoder. Depending on the `reference_size` this read 1, 2, or 4 bytes.

```
protected Any
  readReference;
```

Read an object reference from this decoder and return the object referenced. This raises a `coding-condition` in case not a proper reference is encountered, or if the referenced object is unknown.

```
deferred protected byte
  readByte;
```

Return the next single `byte`.

```
deferred protected void
  readBytes int num
```

```
        to pointer address;
```

Undocumented.

```
protected char
  readChar;
```

Return the next two bytes as a `char`.

```
protected int
  readInt;
```

Return the next four bytes as an `int`.

```
protected long
  readLong;
```

Return the next 8 bytes as a `long`.

File tom/collections

class tom.Collection

inherits

State supers: `State`, `Conditions`, `Copying`, `Enumerable`

instance tom.Collection

methods

```
boolean
  dump_simple_p;
```

Return `YES`.

```
void
  do Block block;
```

Evaluate the `block` for each object element in this `Collection`. Subclasses can provide a faster implementation.

```
boolean
  equal id other;
```

Two collections consider themselves equal if they are the same object or when their elements are equal.

```
void
  freeze;
```

Make the receiving collection immutable. This is irreversible. It is a no-op for immutable collections.

```
Collection
  frozen;
```

Return `self` if this is a non-mutable `Collection`. Otherwise, return a non-mutable collection with the same contents.

```
Any
  member All object;
```

Return the element contained in this collection, which is `equal` to the `object`. The default implementation by `Collection` visits the elements using an enumerator.

```
Any
  memq All object;
```

Like `member`, but the element is identified on reference equality.

```
deferred int
  length;
```

Return the number of elements in this `Collection`.

```
void
  makeElementsPerform Invocation invocation;
```

Fire the `invocation` at the elements contained in the receiving collection.

```
void
  makeElementsPerform selector message;
```

Send the argumentless `message` to the elements contained in the receiving collection.

```
void
  makeElementsPerform selector message
                 with All argument;
```

Send the `message` with the object `argument` to the elements contained in the receiving collection.

```
boolean
  mutable;
```

A `Collection` is not mutable.

```
void
  passElementsTo Invocation inv;
```

Fire the invocation repeatedly, each time with the next object from the collection completing the invocation.

```
OutputStream
  write OutputStream s;
```

Undocumented.

class tom.MutableCollection

inherits

State supers: `Collection`

instance tom.MutableCollection

methods

```
deferred void
  add All object;
```

Undocumented.

```
void
  addElementsFrom Enumerable other;
```

Undocumented.

```
void
  addElementsFromEnumerator Enumerator e;
```

Undocumented.

```
deferred void
  empty;
```

Remove all elements from the receiving collection.

```
deferred void
  freeze;
```

Force `freeze` to be undefined since each particular subclass must itself implement it.

```
id (self)
  initWithEnumerator Enumerator e;
```

Initialize by feeding `self` the elements from the `enumerator`.

```
boolean
  mutable;
```

A `MutableCollection` is mutable.

class tom.Keyed

A `Keyed Collection` stores elements on a key.

inherits

State supers: `Collection`

instance tom.Keyed

methods

```
deferred Any
  at All key
pre
  key != nil;
```

Undocumented.

```
deferred Enumerator
  keyEnumerator;
```

Return an enumerator on the keys of this mapped collection.

```
Any
  member All object;
```

Member for a Keyed collection can be implemented efficiently.

```
Any
  memq All object;
```

Like `member`, but the element is identified on reference equality. This is a less-efficient abstract implementation.

class tom.MutableKeyed

inherits

State supers: `Keyed`, `MutableCollection`

instance tom.MutableKeyed

methods

```
deferred void
  add All object;
```

Add the `object`.

```
deferred void
  remove All object;
```

Remove the `object`.

```
void
  removeElementsFrom Collection c;
```

Remove all the objects contained in the collection `c`.

```
void
  removeElementsFromEnumerator Enumerator e;
```

Undocumented.

```
void
  keepElementsFrom Collection c;
```

Remove all the objects not contained in the collection `c`, i.e., change the receiving collection into the result of intersecting `self` and `c`.

Keyed collections that can not handle losing elements while being enumerated must reimplement this. There is no method `keepElementsFromEnumerator` because set intersection is not meaningful with an arbitrary enumerator.

```
Indexed
  allKeys;
```

All the keys in a convenient format.

class tom.Mapped

A `Mapped Collection` is a `Keyed` collection which stores (key, value) associations.

inherits

State supers: `Keyed`

instance tom.Mapped

methods

```
void
  doKeys Block block;
```

Evaluate the `block` for each key. Subclasses can provide a faster implementation.

```
deferred MapEnumerator
  valueEnumerator;
```

Return a `MapEnumerator` on the values of this `Mapped` collection.

```
boolean
  equal id other;
```

Check for equality, checking not only the values, but also the keys.

```
deferred Enumerator
  keyEnumerator;
```

Return an `Enumerator` on the keys of this `Mapped` collection.

class tom.MutableMapped

inherits

State supers: `Mapped`, `MutableKeyed`

variables

```
const MAPPED_KEEP = 0;
```

Directives to guide `addPairsFrom`.

```
const MAPPED_ERROR = 1;
```

```
const MAPPED_CLOBBER = 2;
```

instance tom.MutableMapped

methods

```
void
  add All value;
```

Add a new pair, using the `value` as the value and the key.

```
deferred void
  set All value
   at All key
pre
  key != nil && value != nil;
```

Undocumented.

```
void
   addPairsFrom Mapped m
```

```
  onContention: int action = MAPPED_KEEP;
```

Add pairs from another `Mapped` collection. The optional `onContention` parameter specifies whether or not pairs which have contending keys should keep the value currently in `self`, overwrite (clobber) the value currently in `self` with the value in `m`, or raise a `type-condition` `Condition`.

```
Indexed
  allValues;
```

Return all values in an `Indexed` format.

class tom.Ordered

An `Ordered` `Collection` maintains its elements in a specific order, though the time complexity or retrieving the nth object not necessarily independent of n.

inherits

State supers: `Collection`

instance tom.Ordered

methods

```
(int, int)
  adjustRange (int, int) (start, len);
```

Adjust the range (`start`, `len`) to fit the length of the receiving `Indexed` collection.

```
Any
  at int index
pre
  index >= 0 && index < [self length];
```

Return the element at `index`. If the receiving collection stores unboxed values, such as integers, the value returned is the element boxed. Returns nil on index overflow (precondition should whine about it, though - nil may only be returned if [self length] is bugged).

```
byte
  at int index;
```

Return the byte value of the element at `index`. If the receiving collection stores objects, the `byteValue` of the element retrieved is actually returned.

```
char
  at int index;
```

The following all follow the *basic type* `at` *index* stanza.

```
int
```

```
  at int index;
```

Undocumented.

```
long
  at int index;
```

Undocumented.

```
float
  at int index;
```

Undocumented.

```
double
  at int index;
```

Undocumented.

```
int
  indexOf All element;
```

Return the index of the first `element`, or -1 if it could not be found.

```
int
  indexOfIdentical All element;
```

Return the index of the first identical `element`, or -1 if it could not be found.

class tom.MutableOrdered

inherits

State supers: `Ordered`, `MutableCollection`

instance tom.MutableOrdered

methods

```
deferred void
  set All object
   at int index;
```

Store the `object` at `index` in the receiving collection.

```
deferred void
  swap (int, int) (i, j);
```

Swap the elements at the indices `i` and `j`.

```
void
  reverse (int, int) (start, len);
```

Reverse the elements in the range starting at `start`, with length `len`.

```
void
  reverse;
```

Reverse the entire collection.

```
deferred void
  removeElementAt int index;
```

Remove the element at `index`, decreasing by 1 the indices of the elements further in the collection.

```
void
  removeElement All element;
```

Remove the first occurence of `element`.

```
void
  removeIdenticalElement All element;
```

Remove the first occurence of the identical `element`.

class tom.Indexed

An `Indexed Collection` maintains an association between integer indices and the objects it contains, with the promise that retrieving an object through the index is O(1) in time complexity.

inherits

State supers: `Ordered`

instance tom.Indexed

methods

```
void
  do Block block;
```

Evaluate the `block` for each object element in this `Indexed`.

```
dynamic
  elements;
```

Extract the elements from the receiving collection, as indicated by the return type. The number of elements in the collection must match the number of expected elements.

```
dynamic
```

```
  elements (int, int) (start, num);
```

Like `elements`, but extract only the `num` elements starting at index `start`.

```
Enumerator
  enumerator;
```

Return an `Enumerator` on this `Indexed`.

```
boolean
  equal id other;
```

A faster implementation than the one inherited from `Collection`.

```
int
  indexOf All element;
```

Return the index of the first `element`, or -1 if it could not be found.

```
int
  indexOfIdentical All element;
```

Return the index of the first identical `element`, or -1 if it could not be found.

```
void
  makeElementsPerform Invocation inv;
```

A faster implementation than the one inherited from `Collection`, without using an `Enumerator`.

```
void
  makeElementsPerform selector message;
```

Likewise.

```
void
  makeElementsPerform selector message
                with All argument;
```

Likewise.

```
void
  passElementsTo Invocation inv;
```

Likewise.

class tom.MutableIndexed

inherits

State supers: `Indexed`, `MutableOrdered`

instance tom.MutableIndexed

class tom.IndexedEnumerator

The `IndexedEnumerator` enumerates any `Indexed` collection, returning the elements boxed.

inherits

State supers: `State`, `Enumerator`

methods

```
instance (id)
  with Indexed indexed;
```

Undocumented.

instance tom.IndexedEnumerator

variables

```
int next;
```

The index of the next element to be returned.

```
int num;
```

The index one beyond the last element to be returned.

```
Indexed indexed;
```

The actual indexed collection.

methods

```
id (self)
    init Indexed a
  start: int start = 0
  length int length;
```

Designated initializer.

```
(boolean, Any)
  next;
```

Undocumented.

```
(boolean, byte)
  next;
```

All these `next` methods are not really necessary, as they are provided by the `Enumerator` behaviour. However, binding them here directly greatly enhances speed and reduces memory requirements for non-object indexeds, which now also do not need their own enumerator to obtain speed.

```
(boolean, char)
  next;
```

Undocumented.

```
(boolean, int)
  next;
```

Undocumented.

```
(boolean, long)
  next;
```

Undocumented.

```
(boolean, float)
  next;
```

Undocumented.

```
(boolean, double)
  next;
```

Undocumented.

File tom/config

class tom.Constants (config)

variables

```
const TOM_RESOURCES = "/usr/local/lib/tom/charmaps/";
```

```
const TOM_PREFIX = "/usr/local/";
```

instance tom.Constants (config)

File tom/holes

class tom.Sink

`Sink` implements an `OutputStream` that is a black hole. It consumes all data that is written to it, and has no value for printing.

inherits

State supers: `State`, `OutputStream`

instance tom.Sink

methods

```
void
  close;
```

Undocumented.

```
void
  write byte b;
```

Undocumented.

```
int
  write byte b;
```

Undocumented.

```
int
  writeRange (int, int) (start, length)
        from ByteArray buffer;
```

Undocumented.

```
id
  print boolean b;
```

Undocumented.

```
id
  print byte b;
```

Undocumented.

```
id
  print char c;
```

Undocumented.

```
id
  print int i;
```

Undocumented.

```
id
  print long l;
```

Undocumented.

```
id
  print float f;
```

Undocumented.

```
id
  print double d;
```

Undocumented.

```
id
  print pointer addr;
```

Undocumented.

```
id
  print dynamic x;
```

Undocumented.

File tom/numbers

class tom.ByteNumber

inherits

State supers: `Number`

instance tom.ByteNumber

variables

```
byte value;
```

methods

```
byte
  byteValue;
```

Undocumented.

```
char
  charValue;
```

Undocumented.

```
double
  doubleValue;
```

Undocumented.

```
float
  floatValue;
```

Undocumented.

```
int
  intValue;
```

Undocumented.

```
long
  longValue;
```

Undocumented.

```
int
  compare Number n;
```

Undocumented.

```
int
  compare byte v;
```

Undocumented.

```
int
  compare char v;
```

Undocumented.

```
int
  compare int v;
```

Undocumented.

```
int
```

```
  compare long v;
```

Undocumented.

```
int
  compare float v;
```

Undocumented.

```
int
  compare double v;
```

Undocumented.

```
id
  init byte value;
```

Undocumented.

```
id
  init char value;
```

Undocumented.

```
id
  init int value;
```

Undocumented.

```
id
  init long value;
```

Undocumented.

```
id
  init float value;
```

Undocumented.

```
id
  init double value;
```

Undocumented.

```
void
  encodeUsingCoder Encoder coder;
```

Undocumented.

```
void
  initWithCoder Decoder coder;
```

Undocumented.

```
OutputStream
  write OutputStream s;
```

Undocumented.

class tom.CharNumber

inherits

State supers: `Number`

instance tom.CharNumber

variables

```
char value;
```

methods

```
byte
  byteValue;
```

Undocumented.

```
char
  charValue;
```

Undocumented.

```
double
  doubleValue;
```

Undocumented.

```
float
  floatValue;
```

Undocumented.

```
int
  intValue;
```

Undocumented.

```
long
  longValue;
```

Undocumented.

```
int
  compare Number n;
```

Undocumented.

```
int
  compare byte v;
```

Undocumented.

```
int
  compare char v;
```

Undocumented.

```
int
  compare int v;
```

Undocumented.

```
int
  compare long v;
```

Undocumented.

```
int
  compare float v;
```

Undocumented.

```
int
  compare double v;
```

Undocumented.

```
id
  init byte value;
```

Undocumented.

```
id
  init char value;
```

Undocumented.

```
id
  init int value;
```

Undocumented.

```
id
  init long value;
```

Undocumented.

```
id
  init float value;
```

Undocumented.

```
id
  init double value;
```

Undocumented.

```
void
  encodeUsingCoder Encoder coder;
```

Undocumented.

```
void
  initWithCoder Decoder coder;
```

Undocumented.

```
OutputStream
  write OutputStream s;
```

Undocumented.

class tom.IntNumber

inherits

State supers: `Number`

instance tom.IntNumber

variables

```
int value;
```

methods

```
byte
  byteValue;
```

Undocumented.

```
char
  charValue;
```

Undocumented.

```
double
  doubleValue;
```

Undocumented.

```
float
  floatValue;
```

Undocumented.

```
int
  intValue;
```

Undocumented.

```
long
  longValue;
```

Undocumented.

```
int
  compare Number n;
```

Undocumented.

```
int
  compare byte v;
```

Undocumented.

```
int
  compare char v;
```

Undocumented.

```
int
  compare int v;
```

Undocumented.

```
int
  compare long v;
```

Undocumented.

```
int
```

```
  compare float v;
```

Undocumented.

```
int
  compare double v;
```

Undocumented.

```
id
  init byte value;
```

Undocumented.

```
id
  init char value;
```

Undocumented.

```
id
  init int value;
```

Undocumented.

```
id
  init long value;
```

Undocumented.

```
id
  init float value;
```

Undocumented.

```
id
  init double value;
```

Undocumented.

```
void
  encodeUsingCoder Encoder coder;
```

Undocumented.

```
void
  initWithCoder Decoder coder;
```

Undocumented.

```
OutputStream
  write OutputStream s;
```

Undocumented.

class tom.LongNumber

inherits

State supers: `Number`

instance tom.LongNumber

variables

```
long value;
```

methods

```
byte
  byteValue;
```

Undocumented.

```
char
  charValue;
```

Undocumented.

```
double
  doubleValue;
```

Undocumented.

```
float
  floatValue;
```

Undocumented.

```
int
  intValue;
```

Undocumented.

```
long
  longValue;
```

Undocumented.

```
int
  compare Number n;
```

Undocumented.

```
int
  compare byte v;
```

Undocumented.

```
int
  compare char v;
```

Undocumented.

```
int
  compare int v;
```

Undocumented.

```
int
  compare long v;
```

Undocumented.

```
int
  compare float v;
```

Undocumented.

```
int
  compare double v;
```

Undocumented.

```
id
  init byte value;
```

Undocumented.

```
id
  init char value;
```

Undocumented.

```
id
  init int value;
```

Undocumented.

```
id
  init long value;
```

Undocumented.

```
id
  init float value;
```

Undocumented.

```
id
  init double value;
```

Undocumented.

```
void
  encodeUsingCoder Encoder coder;
```

Undocumented.

```
void
  initWithCoder Decoder coder;
```

Undocumented.

```
OutputStream
  write OutputStream s;
```

Undocumented.

class tom.FloatNumber

inherits

State supers: `Number`

instance tom.FloatNumber

variables

```
float value;
```

methods

```
byte
  byteValue;
```

Undocumented.

```
char
  charValue;
```

Undocumented.

```
double
  doubleValue;
```

Undocumented.

```
float
  floatValue;
```

Undocumented.

```
int
  intValue;
```

Undocumented.

```
long
  longValue;
```

Undocumented.

```
int
  compare Number n;
```

Undocumented.

```
int
  compare byte v;
```

Undocumented.

```
int
  compare char v;
```

Undocumented.

```
int
  compare int v;
```

Undocumented.

```
int
  compare long v;
```

Undocumented.

```
int
  compare float v;
```

Undocumented.

```
int
```

```
  compare double v;
```

Undocumented.

```
id
  init byte value;
```

Undocumented.

```
id
  init char value;
```

Undocumented.

```
id
  init int value;
```

Undocumented.

```
id
  init long value;
```

Undocumented.

```
id
  init float value;
```

Undocumented.

```
id
  init double value;
```

Undocumented.

```
void
  encodeUsingCoder Encoder coder;
```

Undocumented.

```
void
  initWithCoder Decoder coder;
```

Undocumented.

```
OutputStream
  write OutputStream s;
```

Undocumented.

class tom.DoubleNumber

inherits

State supers: `Number`

instance tom.DoubleNumber

variables

```
double value;
```

methods

```
byte
  byteValue;
```

Undocumented.

```
char
  charValue;
```

Undocumented.

```
double
  doubleValue;
```

Undocumented.

```
float
  floatValue;
```

Undocumented.

```
int
  intValue;
```

Undocumented.

```
long
  longValue;
```

Undocumented.

```
int
  compare Number n;
```

Undocumented.

```
int
  compare byte v;
```

Undocumented.

```
int
  compare char v;
```

Undocumented.

```
int
  compare int v;
```

Undocumented.

```
int
  compare long v;
```

Undocumented.

```
int
  compare float v;
```

Undocumented.

```
int
  compare double v;
```

Undocumented.

```
id
  init byte value;
```

Undocumented.

```
id
  init char value;
```

Undocumented.

```
id
  init int value;
```

Undocumented.

```
id
  init long value;
```

Undocumented.

```
id
  init float value;
```

Undocumented.

```
id
  init double value;
```

Undocumented.

```
void
  encodeUsingCoder Encoder coder;
```

Undocumented.

```
void
  initWithCoder Decoder coder;
```

Undocumented.

```
OutputStream
  write OutputStream s;
```

Undocumented.

File tom/streams

class tom.Stream

inherits

State supers: `Conditions`

Behaviour supers: `All`

instance tom.Stream

inherits

State supers: `Conditions`

Behaviour supers: `All`

methods

```
deferred void
  close;
```

Undocumented.

class tom.InputStream

inherits

State supers: `Stream`

instance tom.InputStream

methods

```
id
  flushInput;
```

Discard any unread input. The default implementation does nothing.

```
deferred byte
  read;
```

Read a byte from the receiving stream. This raises a `stream-eos` condition upon end-of-stream.

```
deferred int
  read;
```

Read a byte from the receiving stream. Return -1 for end-of-stream.

```
int
  readBytes int num
       into MutableByteArray buffer;
```

Read at most `num` bytes from the stream into the `buffer`, starting to add bytes at its current length. Return the number of bytes successfully read, which is 0 for end-of-stream.

```
deferred int
  readRange (int, int) (start, num)
       into MutableByteArray buffer;
```

Read at most `num` bytes from the stream into the `buffer`, by writing in it from position `start`. Return the number of bytes successfully read, which is 0 for end-of-stream.

```
MutableByteString
  readLine;
```

Read a '\n' terminated sequence of bytes and return them (without the '\n' at the end). Return `nil` upon end of file (if no characters have been collected).

```
MutableByteArray
  readLineInto MutableByteArray buf
     truncate: boolean trunc = YES;
```

Read a '\n' terminated sequence of bytes and return them (without the '\n' at the end) in `buf`. Return `nil` upon end of file (if no characters have been collected). If the optional `truncate` is not NO, the buffer is truncated before use.

class tom.OutputStream

inherits

State supers: `Stream`

variables

```
local static MutableByteString print_buffer;
```

The buffer used by `print base:....`

instance tom.OutputStream

methods

```
id
  flushOutput;
```

Write out any unwritten (buffered) output. The default implementation does nothing.

```
id
  nl;
```

Output a new line to the receiving stream. An interactive stream should override this to also flush its output if it desires line based buffering.

```
id
  print boolean b;
```

Undocumented.

```
id
  print byte b;
```

Undocumented.

```
id
  print char c;
```

Undocumented.

```
id
  print int i;
```

Undocumented.

```
id
  print long l;
```

Undocumented.

```
id
  print float f;
```

Undocumented.

```
id
  print double d;
```

Undocumented.

```
id
  print pointer addr;
```

Undocumented.

```
id
  print All object;
```

Undocumented.

```
id
  print dynamic x;
```

Send `print` to `self` for each element of the tuple `x`.

```
id
    print int value
    base: int base = 10
   space: int space = 0
   flush: int flush = -1
  signed: int how_signed = -1
   range: char digit_10 = char ('a');
```

Output the `value` to the receiving stream.

The optional `base` dictates the base of the representation, which defaults to 10.

If the optional `space` is not 0, it is the number of positions the representation must at least occupy.

If `space` is not 0, the optional `flush` dictates how the representation is to be flushed. A negative value means left, 0 for center, and a positive value dictates a right shift. The absolute value of `flush` indicates the amount of whitespace which must be available at the other end.

The optional `signed` should be 0 for unsigned, or 1 for signed. If it is -1 (the default) the value is assumed unsigned, unless `base` has its default value, 10.

The optional `digit_10` sets the value to use for the decimal value 10 when using a `base` exceeding that value.

```
deferred void
  write byte b;
```

Write the byte `b`, signaling a condition upon eof.

```
deferred int
```

```
  write byte b;
```

Write the byte `b` and return the number of bytes actually written.

```
deferred int
  writeBytes int num
        from pointer address;
```

The lowest level multiple-byte writing method: Write the `num` bytes from `address` to the stream, and return the number of bytes written.

```
int
  writeRange (int, int) (start, length)
        from ByteArray buffer;
```

Undocumented.

```
int
  writeBytes ByteArray buffer;
```

Undocumented.

class tom.InputOutputStream

inherits

State supers: `InputStream`, `OutputStream`

instance tom.InputOutputStream

class tom.SeekableStream

inherits

State supers: `Stream`, `Constants`

instance tom.SeekableStream

methods

```
deferred long
  position;
```

Return the current position.

```
deferred void
       seek long offset
  relative: int whence = STREAM_SEEK_SET;
```

Set the position. Any following operation will operate from the new position. The optional argument `whence` defaults to `STREAM_SEEK_SET`, for absolute positioning. Possible other values are `STREAM_SEEK_CUR`, for positioning relative to the current position, and `STREAM_SEEK_END`, to work relative to the end.

class tom.StreamStream

inherits

State supers: `Stream`, `State`

methods

```
instance (id)
  with Stream s;
```

Undocumented.

instance tom.StreamStream

variables

```
public InputOutputStream stream;
```

The stream to which we output and/or from which we input.

methods

```
void
  close;
```

Undocumented.

```
id
  init Stream s;
```

Undocumented.

class tom.stdio

inherits

Behaviour supers: `All`

variables

```
static public InputStream in;
```

The stream connected to descriptor 0, known as `stdin` in C.

```
static public OutputStream out;
```

The stream connected to descriptor 1, C's `stdout`.

```
static public InputOutputStream err;
```

The stream connected to descriptor 2, C's `stderr`. Like `out`, this stream is buffered.

methods

```
void
  close int descriptor;
```

Close the `descriptor`. Raises a `stream-error` on failure.

instance tom.stdio

inherits

Behaviour supers: `All`

File tom/unique-strings

class tom.UniqueString

inherits

State supers: `String`

variables

```
static MutableSet strings;
```

The `Container` of all unique strings. The container mechanism will ensure only those strings are kept that are still needed.

methods

```
void
  load Array arguments;
```

Initialize the `strings` `Container`.

```
instance (id)
  with String s;
```

Return the `UniqueString` containing the same information as the `String` `s`. This is the only method to be used to create unique strings.

```
protected instance (id)
```

```
  awake instance (id) us
  equal selector cmp;
```

Find the string `us` in the `strings`, comparing them using the selector `cmp`. If the string is found, the old string is returned. Otherwise, `us` is added to the `strings` and it is returned.

instance tom.UniqueString

variables

```
int hash;
```

The cached hash value of this string.

methods

```
boolean
  equal String other;
```

Tell the `other` to compare itself to this `UniqueString`.

```
boolean
  equalUniqueString UniqueString other;
```

Return `TRUE`. This is simple pointer comparison.

```
int
  compare id other;
```

Override compare, to return fast upon equality.

class tom.UniqueByteString

inherits

State supers: `UniqueString`, `ByteString`

instance tom.UniqueByteString

methods

```
boolean
  equal String other;
```

This definition is only here because we can not direct the implementation of this selector to the right class.

```
boolean
  equalUniqueString UniqueString other;
```

The same is true for this one.

```
int
  hash;
```

When called for the first time, hash the string, and remember the value. Every next time, return the cached value. Obviously, this loses for strings with a zero hash value (which keep being hashed).

```
id
  awakeAfterUsingCoder Decoder coder;
```

Iff a unique string like the receiving one already exists, return the already existing one. Otherwise, add the receiving string to the known unique strings.

class tom.UniqueCharString

inherits

State supers: `UniqueString`, `CharString`

instance tom.UniqueCharString

methods

```
boolean
  equal String other;
```

Redefinition since we can not redirect to the right (`UniqueString`) super.

```
boolean
  equalUniqueString UniqueString other;
```

Redefinition since we can not redirect to the right (`UniqueString`) super.

```
int
  hash;
```

Cached hashing, just like `UniqueByteString`.

```
id
  awakeAfterUsingCoder Decoder coder;
```

Iff a unique string like the receiving one already exists, return the already existing one. Otherwise, add the receiving string to the known unique strings.

Chapter 9. Unit `C`

The `C` unit interfaces TOM with `libc` and `libm`.

File C/Math

The `Math` class wraps the standard mathematical functions present within libc.

class C.Math

Return the length of the hypotenuse of a right triangle whose sides measure `x` and `y` in length.

methods

```
double
  hypot (double, double) (x, y);
```

Undocumented.

```
double
  lgamma double x;
```

Undocumented.

```
double
  erf double x;
```

Undocumented.

```
double
  erfc double x;
```

Undocumented.

```
double
  j0 double x;
```

Undocumented.

```
double
  j1 double x;
```

Undocumented.

```
double
  jn (int, double) (x, n);
```

Undocumented.

```
double
  y0 double x;
```

Undocumented.

```
double
  y1 double x;
```

Undocumented.

```
double
  yn (int, double) (x, n);
```

Undocumented.

```
double
  acos double x;
```

Return the arc cosine of `x`. `x` must not fall outside the range -1 to 1. The return value will be in radians.

```
double
  asin double x;
```

Return the arc sine of `x`. `x` must not fall outside the range -1 to 1. The return value will be in radians.

```
double
  atan double x;
```

Return the arc tangent of `x`. The return value will be in radians.

```
double
  atan2 (double, double) (x, y);
```

Return the arc tangent of `y/x`, using the signs of the arguments to determine the quadrant of the return value. The return value will be in radians.

```
double
  cos double x;
```

Return the cosine of `x`. The return value will be in radians

```
double
  sin double x;
```

Return the sine of `x`. The return value will be in radians.

```
double
  tan double x;
```

Return the tangent of `x`. The return value will be in radians.

```
double
  cosh double x;
```

Return the hyperbolic cosine of x. The return value will be in radians.

```
double
  sinh double x;
```

Return the hyperbolic sine of x. The return value will be in radians.

```
double
  tanh double x;
```

Return the hyperbolic tangent of x. The return value will be in radians.

```
double
  exp double x;
```

Return the result of computing e^x.

```
double
  ldexp (double, int) (x, n);
```

Return the result of computing x * 2^n.

```
double
  log double x;
```

Return the natural log of x. x must be positive.

```
double
  log10 double x;
```

Return the log of x in base 10. x must be positive.

```
double
  pow (double, double) (x, y);
```

Return the value of x raised to the power y, x^y. If x is negative, y must be an integer value.

```
double
  sqrt double x;
```

Return the square root of x.

```
double
  ceil double x;
```

Return the value of x rounded up to the nearest integer.

```
double
```

```
  fabs double x;
```

Return the absolute value of `x`.

```
double
  floor double x;
```

Return the value of `x` rounded down to the nearest integer.

```
double
  fmod (double, double) (x, y);
```

Return the remainder of dividing `x` by `y`.

instance C.Math

File C/Std

This class wraps functions from ‘stdio.h’, ‘stdlib.h’, and other ANSI C headers.

class C.Std

Immediately abort execution of the program, causing a core dump on some systems.

methods

```
void
  abort;
```

Undocumented.

```
void
  sleep int seconds;
```

Suspend execution for `seconds` seconds.

```
int (result)
  system String cmdline;
```

Execute `cmdline` via the system() system call. All limitations of the underlying system() call apply.

```
int
  abs int intval;
```

Compute the absolute value of the integer `intval`.

instance C.Std

Chapter 10. Unit `too`

The `too` unit provides TOM with networking, event dispatching, and distributed objects.

File too/AutoLock

class tom.Conditions (AutoLock)

This extension provides deadlock-condition.

variables

```
static ConditionClass deadlock-condition;
```

instance tom.Conditions (AutoLock)

class tom.Thread (AutoLock)

Necessary glue within `Thread` class.

instance tom.Thread (AutoLock)

variables

```
public Thread blockedBy;
```

The thread which has to release a lock so we can continue.

methods

```
boolean
  isWaitingFor Thread t;
```

Return `TRUE` if we are waiting for the argument `Thread t` to finish. This includes implied (transitive) waiting.

```
void
  setBlockedBy Thread t;
```

Set `blockedBy`. This is solely used internally by the `AutoLock` class.

class too.AutoLock

Recursive lock with deadlock detection. If deadlock occurs, the faulty thread will be unjammed by receiving `deadlock-condition`.

inherits

State supers: `RecursiveLock`, `Conditions`

methods

```
void
  load MutableArray arguments;
```

Initialization method.

instance too.AutoLock

variables

```
Thread owner;
```

The thread holding the lock, `nil` if none.

methods

```
id
  init;
```

Designated initializer.

```
void
  lock;
```

Obtain the lock, but if a deadlock is detected, `deadlock-condition` will be raised.

```
void
  unlock;
```

Unlock the lock.

File too/Connection

class too.Connection

inherits

State supers: `State`, `DescriptorReadDelegate`, `Constants`

variables

```
static MutableEqSet all_connections;
```

All connection objects.

methods

```
instance (id)
  alloc;
```

Store the new connection in the `all_connections`.

```
void
       connection Connection connection
  remoteProxyDead int identity
pre
  !!all_connections[connection];
```

Pass this message to the `connection`.

instance too.Connection

variables

```
public Any root;
```

The root object of this connection.

```
public Port port;
```

The Port serving this connection.

```
MutableEqDictionary local_objects;
```

The set of local proxies, keyed on their local object.

```
MutableIntDictionary local_proxies;
```

The set of local proxies, keyed on their identity.

```
MutableIntDictionary remote_proxies;
```

The set of remote proxies, keyed on their identity.

```
MutableIntArray unreported_deaths;
```

The set of remote proxy identities that are dead here and which need to be sent to the other side.

```
int last_proxy_ident;
```

The last number used as a local proxy identity.

methods

```
id (self)
```

```
  initWithPort Port p;
```

Designated initializer.

```
protected IntDictionary
  local_proxies;
```

Other connections may inspect our proxies.

```
Any (object)
  localObject int identity
post
  object != nil;
```

Return the local object identified by the `identity` to the other side.

```
Proxy
  localProxyFor All object
pre
  object != nil;
```

Return the local proxy to identify the local `object`.

```
Any
  remoteObject int identity;
```

Return the remote object identified by the `identity` by the other side.

```
void
  localProxyRelease int identity
pre
  !!local_proxies[identity];
```

Be informed that the local proxy with the `identity` has one less remote proxy to care for. If that number reaches zero, the local proxy object is removed.

```
void
  remoteProxyDead int identity;
```

By informed (by our remote proxy with the `identity`) of the GC death of a remote proxy.

Note that this method is invoked during GC and that no new objects should be allocated.

class too.ServerConnection

inherits

State supers: `Connection`

instance too.ServerConnection

variables

```
redeclare ServerPort port;
```

Our `port` is only here for accepting connections.

methods

```
id (self)
  initWithPort ServerPort p;
```

Designated initializer.

```
void
  set_root All r;
```

Undocumented.

```
void
  readEventOnDescriptor ServerInetPort p;
```

Instantiate another `ConnectedConnection`.

class too.ConnectedConnection

inherits

State supers: `Connection`, `DescriptorWriteDelegate`, `Conditions`

instance too.ConnectedConnection

variables

```
redeclare ConnectedPort port;
```

We are actually connected.

```
PortDecoder decoder;
```

Our decoder.

```
PortEncoder encoder;
```

Our encoder.

```
ServerConnection master;
```

If we're a slave connection (i.e. the working part for a published connection), this is the published server connection.

```
boolean invalid;
```

Iff `TRUE`, we've lost the connection.

methods

```
id (self)
  initWithPort ConnectedPort p;
```

Initializer for a client connection.

```
id (self)
  initWithPort ConnectedPort p
           for ServerConnection server;
```

Initializer for a slave connection, i.e. a slave to the `server` connection.

```
protected void
  initDetails;
```

Do part of the work for either initializer.

```
void
  invalidate;
```

Undocumented.

```
ConnectedPort
  port;
```

Undocumented.

```
Any
  localObject int identity;
```

Forward to the `master` if we have one.

```
Proxy
  localProxyFor All object;
```

Forward to the `master` if we have one.

```
void
  localProxyRelease int identity
pre
  !!master -> !![master local_proxies][identity];
```

Forward to the `master` if we have one.

```
InvocationResult
  forward Invocation invocation;
```

Forward the `invocation` to the other side.

```
void
  readEventOnDescriptor ConnectedInetPort p;
```

Undocumented.

File too/DescriptorDelegate

class too.DescriptorDelegate

DescriptorDelegate classes are used to define the logic for handling read and write events on file `Descriptors`. Users should create new classes inheriting from either `DescriptorReadDelegate` or `DescriptorWriteDelegate`, providing an implementation of either `readEventOnDescriptor` or `writeEventOnDescriptor` respectively, to implement their application logic.

instance too.DescriptorDelegate

class too.DescriptorReadDelegate

inherits

State supers: `DescriptorDelegate`

instance too.DescriptorReadDelegate

methods

```
deferred void
  readEventOnDescriptor Descriptor d;
```

The `RunLoop` has determined that the `Descriptor d` is readable.

class too.DescriptorWriteDelegate

inherits

State supers: `DescriptorDelegate`

instance too.DescriptorWriteDelegate

methods

```
deferred void
  writeEventOnDescriptor Descriptor d;
```

The `RunLoop` has determined that the `Descriptor d` is writable.

File too/DescriptorSet

class too.DescriptorSet

inherits

State supers: `State`, `C`

instance too.DescriptorSet

variables

`pointer set;`

The bitset of descriptors usable to select(2).

`int cap;`

One beyond the highest descriptor that can be put in the set.

`int beyond_last;`

One beyond the highest descriptor present in the set.

`int num;`

The number of descriptors present in the set.

`MutableObjectArray descriptors;`

The array of `Descriptor` objects.

`MutableObjectArray delegates;`

The array of `DescriptorDelegate` objects.

methods

```
void
  dealloc;
```

Deallocate the memory occupied by the `set`.

```
id (self)
  init;
```

Designated initializer.

```
void
  remove Descriptor descriptor;
```

Undocumented.

```
void
  set DescriptorDelegate delegate
   at Descriptor descriptor;
```

Undocumented.

```
(pointer, int)
  vitals;
```

Return a pointer to the low-level descriptor `set`, and the one beyond the highest value in that set.

```
void
  readEvent int d;
```

Dispatch a read event on the file descriptor `d`, to the delegate at index `d` in the delegates.

```
void
  writeEvent int d;
```

Dispatch a write event on the file descriptor `d`, to the delegate at index `d` in the delegates.

File too/PortCoder

class tom.Encoder (PortCoder)

instance tom.Encoder (PortCoder)

methods

```
boolean
  encodeProxy All p;
```

Encode a `Proxy`. If this is for archiving purposes, this does nothing and returns `FALSE` (the default implementation). Otherwise, in case of wiring, it actually performs the proxy encoding and returns `TRUE`.

class too.PortCoder

inherits

State supers: `BinaryCoder`

instance too.PortCoder

variables

```
public ConnectedConnection connection;
```

The `Connection` for which we operate.

```
ConnectedPort port;
```

Our buffered view of the socket in the direction we handle.

methods

```
id
  initWithConnection ConnectedConnection c;
```

Designated initializer.

File too/PortDecoder

class too.PortDecoder

inherits

State supers: `BinaryDecoder`, `PortCoder`

instance too.PortDecoder

methods

```
id
  initWithConnection ConnectedConnection c;
```

Undocumented.

```
Any
  decode byte b;
```

Handle proxy tags before super.

```
protected byte
  readByte;
```

Undocumented.

```
protected void
  readBytes int num
         to pointer address;
```

Undocumented.

File too/PortEncoder

class too.PortEncoder

inherits

State supers: `BinaryEncoder`, `PortCoder`

instance too.PortEncoder

methods

```
id
  initWithConnection ConnectedConnection c;
```

Designated initializer.

```
protected State
  replacementObjectFor State object;
```

Return the object returned by asking `replacementForPortCoder` to the argument `object`.

```
boolean
  encodeProxy Proxy p;
```

Encode the proxy `p` and return `TRUE`.

```
void
  flushOutput;
```

Forward to the `port`.

```
void
  reportDeaths IntArray deaths;
```

Undocumented.

```
protected void
  writeByte byte b;
```

Undocumented.

```
protected void
  writeBytes (int, int) (start, length)
        from ByteArray r;
```

Undocumented.

```
protected void
  writeBytes (pointer, int) (address, length);
```

Undocumented.

File too/Proxy

class tom.State (Proxy)

This extension of `State` only provides the `isProxy` method, which allows one to discern between proxy and non-proxy objects.

instance tom.State (Proxy)

methods

```
boolean
  isProxy;
```

Undocumented.

```
State
  replacementForPortCoder PortEncoder coder;
```

Return the object to be encoded by the `coder` instead of the receiving object. This method is repeatedly invoked until an object returns `self`. The default implementation retrieves a proxy from the `coder`'s `connection`.

class too.Proxy

inherits

State supers: `State`

instance too.Proxy

variables

```
Connection connection;
```

The `Connection` to which we belong.

```
int identity;
```

Our identity with our `connection`.

methods

```
id
  initWithConnection Connection c
            identity int ident;
```

Undocumented.

```
Connection
  proxy_connection;
```

Undocumented.

```
int
  proxy_identity;
```

Undocumented.

```
State
  replacementForPortCoder PortEncoder coder;
```

Return `self`, since we know how to be sent over the wire.

```
void
  encodeUsingCoder Encoder coder;
```

Have the `coder` encode us as a proxy; otherwise fail (which is the case when archiving instead of wiring).

class too.LocalProxy

inherits

State supers: `Proxy`

instance too.LocalProxy

variables

```
public Any original;
```

The object for which we stand.

methods

```
id
  initWithConnection Connection c
            identity int i
                 for All object;
```

Designated initializer.

class too.RemoteProxy

inherits

State supers: `Proxy`

instance too.RemoteProxy

variables

```
redeclare ConnectedConnection connection;
```

Our `connection` is connected.

methods

```
boolean
  isProxy;
```

Undocumented.

```
InvocationResult
  forwardSelector selector sel
        arguments pointer args;
```

The low-level forwarding method. This method is invoked for forwarding a invocation completing method and this is used by the `Proxy`.

```
void
  dealloc;
```

Inform our `connection` from our death. This messages the `Connection` class, since messaging objects from `dealloc` methods is not allowed. We identify ourselves by our `identity` since passing around a dead object (which we are) is asking for trouble.

class too.NonProxy

Instances of (subclasses of) `NonProxy` are never proxies. They always send a copy over the wire.

inherits

State supers: `State`

instance too.NonProxy

methods

```
id (self)
```

`replacementForPortCoder PortEncoder c;`

Return `self` as we do not want to be proxied.

class tom.Number (Proxy)

inherits

State supers: `NonProxy`

instance tom.Number (Proxy)

class tom.Invocation (Proxy)

inherits

State supers: `NonProxy`

instance tom.Invocation (Proxy)

class tom.InvocationResult (Proxy)

inherits

State supers: `NonProxy`

instance tom.InvocationResult (Proxy)

class tom.Selector (Proxy)

inherits

State supers: `NonProxy`

instance tom.Selector (Proxy)

class tom.Collection (Proxy)

inherits

State supers: `NonProxy`

instance tom.Collection (Proxy)

class tom.MutableCollection (Proxy)

inherits

State supers: `State`

instance tom.MutableCollection (Proxy)

methods

```
id
  replacementForPortCoder PortEncoder c;
```

This is naughty: a `Collection`, through its inheritance of `NonProxy` returns `self` when asked its `replacementForPortCoder`. However, a `MutableCollection` must be proxied for maintaining the right semantics. Hence, we redirect the method to our direct (though repeated) superclass, `State`.

File too/RunLoop

class too.RunLoop

inherits

State supers: `State`

variables

```
local static instance (id) current;
```

This thread's run loop.

methods

```
instance (id)
  current;
```

Return this thread's `RunLoop`, creating it if it does not yet exist.

instance too.RunLoop

variables

```
DescriptorSet read_set;
```

The read and write sets.

```
DescriptorSet write_set;
```

```
Heap timers;
```

The timers scheduled with us.

```
public mutable RunLoopDelegate delegate;
```

The delegate, if we have one.

```
boolean d_changed;
```

Iff `TRUE`, one of the descriptor sets was changed, indicating to the `run` method that it should update some of its local variables.

```
boolean t_changed;
```

Iff `TRUE`, the `timers` was changed.

methods

```
id (self)
  init;
```

Designated initializer.

```
void
  run;
```

Run this runloop. This method does not return.

```
void
  addDescriptorForRead Descriptor descriptor
              delegate DescriptorReadDelegate delegate;
```

Add the `descriptor` to this runloop, read events on which are to be handled by the `delegate`. This does not protect against adding the `descriptor` to only a single runloop.

```
void
  addDescriptorForWrite Descriptor descriptor
               delegate DescriptorWriteDelegate delegate;
```

Similar to `addDescriptorForRead delegate`, add the `descriptor` to this runloop, write events on which are to be handled by the `delegate`.

```
void
  removeReadDescriptor Descriptor descriptor;
```

Remove the `descriptor` from this runloop. No check is performed on whether the `descriptor` actually is registered for reading with this runloop.

```
void
  removeWriteDescriptor Descriptor descriptor;
```

Similar to `removeReadDescriptor`, but the `descriptor` is removed from the write set.

```
void
  add_timer Timer timer;
```

Add the `timer` to the current run loop.

```
void
  remove_timer Timer timer;
```

Remove the `timer` which is scheduled with this run loop.

class too.RunLoopDelegate

instance too.RunLoopDelegate

methods

```
deferred void
  runLoopWillSelect RunLoop loop;
```

Be notified that the `RunLoop` `loop` will do another select.

class tom.All (RunLoop)

This extension of `All` provides delayed `perform`ance.

instance tom.All (RunLoop)

methods

```
void
  perform selector sel
    after double seconds
     with dynamic arguments
pre
  seconds >= 0.0;
```

Perform the selector `sel` with the `arguments` after `seconds` delay. Even if `seconds` is 0, the invocation is not fired immediately; a timer is always set to have the `RunLoop` fire the invocation.

File too/Timer

class too.Timer

Instances of the `Timer` class provide, in conjunction with the `RunLoop`, event scheduling functionality. `Timer` objects can fire once or repeatedly.

Because the trigger time of a `Timer` can change, in case of repeated firing, a `Timer` is not a `Date`: a `Timer` represents a moment in time like a `Date`, but a `Date` is assumed to be constant.

inherits

State supers: `HeapElement`

methods

```
instance (id)
  withInterval double secs
    invocation Invocation invocation
      repeats: boolean repeats_p = NO
pre
  secs > 0.0 || !repeats_p && !secs;
```

Return a newly allocated `Timer` instance to fire `secs` from now, with the `Invocation` `invocation`. Iff `repeats_p`, the timer will repeat every `secs`.

instance too.Timer

variables

```
public double fire_time;
```

The next (relative) moment in time we will fire.

```
public double period;
```

The repetition period. This is 0.0 for a single-shot timer.

```
Invocation invocation;
```

The invocation to fire when we do.

methods

```
id (self)
  initWithFireTime double d
        invocation Invocation i
           period: double p = 0.0
pre
  p >= 0.0;
```

Designated initializer. If the time `d` lies in the past, the timer will fire as soon as possible.

```
void
  fire;
```

Fire the timer and invoke the invocation. If the timer is repeating, and the invocation did not throw any conditions, then the timer will be re-added to the current `RunLoop`. Unfortunately, this will fail silently.

```
OutputStream (s)
  writeFields OutputStream s;
```

Undocumented.

```
void
  cancel
pre
  [self scheduled];
```

Cancel this timer with the current `RunLoop`. It must be scheduled with that `RunLoop`.

```
void
  schedule
pre
  ![self scheduled];
```

Schedule this timer with the current `RunLoop`. The timer must not already be scheduled.

```
boolean
  scheduled;
```

Return whether this timer is currently scheduled.

```
int
  compare id other;
```

Return a comparison of the firing times of the two timers.

File too/inet

class too.InetAddress

An `InetAddress` really is an IPv4 address. It depends on the underlying IPv6 implementation if this class is usable for IPv6 addresses.

inherits

State supers: `State`, `Address`

methods

```
instance (id)
  with (InetHost, pointer, int) (h, a, l);
```

Return a new instance with the indicated fields. (The `address` will be deallocated upon the death of the newly created address.)

instance too.InetAddress

variables

`InetHost host;`

The host on which this address resides.

`pointer address;`

The internet address.

`int address_length;`

The length in bytes of the address.

methods

```
(pointer, int)
  osAddress;
```

Return the low-level bare address.

```
void
  dealloc;
```

Undocumented.

```
boolean
  equal id other;
```

Undocumented.

```
int
  hash;
```

Undocumented.

```
InetHost
  host;
```

Return the host of this address. The host is looked up if the address was not yet related to a host.

```
protected id
  init (InetHost, pointer, int) (h, a, l);
```

Designated initializer.

```
OutputStream
  write OutputStream s;
```

Output the address in dotted decimal octet notation.

class too.InetHost

inherits

State supers: `State`, `Host`

variables

```
static MutableDictionary hosts_by_name;
```

All internet hosts currently known, keyed on their name(s).

```
static MutableDictionary hosts_by_addr;
```

All internet hosts currently known, keyed on their address(es).

```
static InetHost local_host_any;
```

The wildcard local host.

methods

```
instance (id)
  addressed InetAddress addr;
```

Return the host known with the address `addr`. If the host can be found in the cache, no lookup is performed.

```
protected void
  cacheHost instance (id) h;
```

Add the host `h` to the cache.

```
void
  initialize;
```

Undocumented.

```
void
  load Array arguments;
```

Undocumented.

```
instance (id)
  named String name;
```

Return the host named `name`. If the host can be found in the cache, no lookup is performed.

```
protected instance (id)
  hostWithAddress InetAddress addr;
```

Perform a lookup of the host addressed `addr`. Return the host, or `nil` if it could not be found. The cache remains unaffected.

```
protected instance (id)
  hostWithName String name;
```

Perform a lookup of the host named `name`. Return the host, or `nil` if it could not be found. The cache remains unaffected.

```
protected instance (id)
  with (Array, Array) (n, a);
```

Return a newly allocated host with the names `n` and addresses `a`.

instance too.InetHost

variables

```
public Array names;
```

The names by which this host is known.

```
public Array addresses;
```

The addresses by which this host is known.

methods

```
id
  init (Array, Array) (n, a);
```

Designated initializer.

```
String
  name;
```

Undocumented.

class too.InetPort

An `InetPort` is an abstract port on an internet host.

inherits

State supers: `Port`, `State`

methods

```
instance (id)
  with int port
    at InetAddress address;
```

Return a newly created `InetPort` with the `address` and the `port`.

instance too.InetPort

variables

```
public InetAddress address;
```

The address (of the host) at which this port resides.

```
public int port;
```

The port on the host.

methods

```
boolean
  equal id other;
```

Undocumented.

```
int
  hash;
```

Undocumented.

```
protected id
  initWithPort int p
            at InetAddress a;
```

Designated initializer.

```
OutputStream
  write OutputStream s;
```

Output the address in dotted decimal octet notation followed by a colon and the port number.

class too.ConnectedInetPort

A `ConnectedInetPort` is a bytestream on a connected TCP socket.

inherits

State supers: `ConnectedPort`, `InetPort`

instance too.ConnectedInetPort

variables

```
public InetPort server;
```

Description of the server to which we connected or from which we accepted. In the latter case, this will be a `ServerInetPort`.

```
InetPort peer;
```

The peer socket. Set by invoking `InetPort [self peer]`.

methods

```
protected id
  initWithPort int p
            at InetAddress a;
```

Designated initializer. Connect to the port `p` at the address `a`.

```
id
  initWithPort int p
            at InetAddress a
    descriptor int d
        server ServerInetPort s
          peer InetPort pr;
```

Initialization used by ‘ConnectedInetPort [ServerInetPort accept]’.

```
InetPort
  peer;
```

Return the peer port.

```
void
  registerForRead DescriptorReadDelegate d;
```

Undocumented.

```
void
  registerForWrite DescriptorWriteDelegate d;
```

Undocumented.

class too.ServerInetPort

A `ServerInetPort` is a TCP port which is listening for connections to accept.

inherits

State supers: `InetPort`, `ServerPort`, `Descriptor`

instance too.ServerInetPort

methods

```
ConnectedInetPort
  accept;
```

Accept a connection on the receiving port, returning a connected port.

```
protected id
  initWithPort int port
            at InetAddress address;
```

Designated initializer. Listen on the `port` at the `address`. If the `address` is `nil`, any local address will do; if the port is 0, it is assigned by the operating system.

```
void
  registerForRead DescriptorReadDelegate d;
```

Undocumented.

File too/network

class too.Address

instance too.Address

methods

```
deferred Host
  host;
```

Return the host on which this address resides.

class too.Host

instance too.Host

methods

```
deferred Array
  names;
```

Return the names for this host.

```
deferred String
  name;
```

Return the canonical name of this host.

File too/ports

class too.Port

instance too.Port

methods

```
deferred void
  registerForRead DescriptorReadDelegate d;
```

Undocumented.

class too.ConnectedPort

inherits

State supers: `Port`, `ByteStream`

instance too.ConnectedPort

methods

```
deferred void
  registerForWrite DescriptorWriteDelegate d;
```

Undocumented.

class too.ServerPort

inherits

State supers: `Port`

instance too.ServerPort

methods

```
deferred ConnectedPort
  accept;
```

Undocumented.

File too/Nameserver

class too.NameserverDefinitions

The NameserverDefinitions class contains nameserver related constants for any interested class to inherit.

variables

```
const DEFAULT_SERVER_PORT = 2360;
```

The default TCP port on which the nameserver is listening.

```
const PORT_NOT_FOUND = -1;
```

The port number returned for the port which is not found.

instance too.NameserverDefinitions

class too.Nameserver

inherits

State supers: `NameserverDefinitions`

instance too.Nameserver

methods

```
deferred void
       reportTo NameserverClient client
  portOfService String service_name
         onHost String hostname;
```

Report to the `client` the internet TCP port of the service named `service_name` which is running on the host named `hostname`.

class too.NameserverClient

inherits

State supers: `NameserverDefinitions`

instance too.NameserverClient

methods

```
deferred void
  service String service_name
   onHost String hostname
  hasPort int port;
```

Be informed of the `port` in response to the `reportTo portOfService onHost` request.

Chapter 11. Unit `_builtin_`

`Any`

The `Any` class...

`Any`

The `Any` instance...

III. Reference

I. Tools man pages

tesla

Name

`tesla` — TOM Compiler

Synopsis

tesla [-u *`unitname`* | -f *`file`*] [*`options`*]

tesla {--version}

Description

tesla is the TOM compiler. It translates the TOM source files into C, which may subsequently be translated by the GNU C compiler, gcc(1), to produce an object file. The input file is usually a unit file, specified by `-u` *`unitname`*. This unit file describes the files which are to compiled by **tesla**. Alternatively, **tesla** may compile a single source file, specified by `-f` *`file`*.

Regular Options

`-I` *`dir`*

Add *`dir`* to the include path.

`-1`

Use the classic, fully dynamic behavior. This will become useful once **tesla** can do whole program compilation for greater optiomization.

`-F`

Write out all files, whether or not they are already present.

`-o` *`file`*

Only output the *`file`* and the header.

`-v`

Be verbose.

`--version`

Report the version of **tug** and exit.

Advanced Options

`-C` *`name`*

Add the class *`name`* to the classes in the *`file`* specified by `-f`.

`-C` *`a:b`*

Like `-C` *`name`*, except make *`A`* pose as *`B`*.

`-E` *`x:y`*

Add the extension *`y`* of the class *`x`* to the *`file`* specified by `-f`.

`--c-extension` *`foo`*

Use '`.`*`foo`*' as the extension of generated source files.

`-a`

Don't depend upon the presence of unit tom.

`-W`*`x`*
`-Wno-`*`x`*

Warn or do not warn about '*`x`*'. Possible warnings are `empty-compound`.

`-f`*`x`*
`-fno-`*`x`*

Include or exclude the *`x`* feature. Known features are `readable-c`.

Debugging Options

`-ve`

Print top expressions while they are read.

`-vm`

Print methods after having been resolved.

`-vf`

Emit names of files that are being read.

`-vF`

Emit names of files that are being resolved.

`:trace-parser tesla.ParseTom`

Tell the parser to output full debug information. This is really a debugging option for the parser and will output a lot of data.

tig

Name

`tig` — TOM Interface Generator

Synopsis

tig {-u *`unitname`*} [-F] [-I *`dir`*] [-v]

tig {--version}

Description

tig is the TOM Interface Generator. It generates the interface definition files for the specified *`unitname`*. These interface definition files contain descriptions of the classes, methods and object variables that comprise that unit. A file is created for every original TOM source file in the unit *`unitname`*, with the extension '`.j`'.

Regular Options

`-I` *`dir`*

Add *`dir`* to the include path.

`-F`

Output all files, whether or not they are already present.

`-v`

Be verbose.

`--version`

Report the version of **tig** and exit.

tug

Name

`tug` — TOM Unit Generator

Synopsis

tug {-u *`unitname`*} {*`files`*...} [-U *`unit`*]

tug {--version}

Description

tug is the TOM Unit Generator. It outputs a unit file, *`unitname`*.u, detailing which *`files`* and classes make up the unit *`unitname`*.

Regular Options

`-U` *`unit`*

Make the unit *`unitname`* depend upon the unit *`unit`*.

`-v`

Be verbose.

`--version`

Report the version of **tug** and exit.

gp

Name

`gp` — TOM Parser Generator

Synopsis

gp [-d] [-v] [-o *`tom_source_file`*] [-o *`tom_defines_file`*] {*`file`*}

gp {--version}

Description

gp is the TOM Parser Generator. It outputs a generated TOM source file, *`tom_source_file`*.t and a secondary TOM source file, *`tom_defines_file`*.t, when given a grammar specification *`file`*.

Regular Options

`-o` *`tom_source_file`*

Specify the name of hte file to be used for the source of the generated parser.

`-o` *`tom_defines_file`*

Specify the name of the file to be used when generating the defines used by the generated parser.

`-d`

Output debugging information while parsing the grammar file..

`-v`

Be verbose.

`-v-flat`

Emit flat rules.

`-v-cost`

Emit non-terminal insertion cost table.

`--version`

Report the version of **gp** and exit. This is not yet implemented.

tomc

Name

`tomc` — TOM compiler (older, being replaced by tesla)

Synopsis

tomc [*options*] [*file*] [*output_c* | *output_info*]

Description

tomc is the TOM compiler. It translates the file file containing TOM source, and outputs a file output_c which can subsequently be translated by the GNU C compiler, gcc(1), to produce an object file. The input file usually has the extension .t. tomc also produces a file output_info, called the info file, which contains information needed by the TOM resolver, tomr(1). If one or both of the output files is omitted, the default name is the basename of the input file, with the extension .c or .i, respectively.

tomc is being retired in favor of **tesla**.

IV. Appendices

Appendix A. TOM makefiles

Basics

The *TOM makefiles* is a collection of makefiles that enable easy building and rebuilding of TOM program units, library units, and dynamically loadable units. When using the TOM makefiles, you do not need to be concerned with the details of compilation or of the particular system you are using, or will be using in the future.

The following makefiles constitute the TOM makefiles:

`GNUmakefile.app`

Build a program into which dynamic loading is possible.

`GNUmakefile.bin`

Build a program into which dynamic loading is not necessarily possible. It depends on the operating system being used whether `GNUmakefile.app` and `GNUmakefile.bin` actualy create different executables (on NeXTSTEP, for instance, they do not differ).

`GNUmakefile.lib`

Build a library unit.

`GNUmakefile.load`

Build a unit which is to be dynamically loaded.

`GNUmakefile.top`

Build only subprojects.

`GNUmakefile.common`

The heart of the TOM makefiles. The others are just front-ends to this file.

This is what a minimal, example `GNUmakefile` looks like:

```
UNIT=               hello
TOM_SRC=            hello

TOM_MAKEFILES_DIR=  /usr/lib/tom/makefiles
include $(TOM_MAKEFILES_DIR)/GNUmakefile.bin
```

To start with the last line, we see the `GNUmakefile.bin` being included. This means the `GNUmakefile` is for a program. The `UNIT` is called *hello*; this will also be the name of the resulting program. This simple program contains only one TOM source file, `hello.t`. Note that the extension is not specified.

Important macros

The following macro's are mandatory except when using the `GNUmakefile.top`.

`UNIT`

The name of the unit being built. It depends on the actual makefile being used whether the unit will be built as a library, application, etc.

`TOM_SRC`

The names of the TOM source files in this unit, without extension. All TOM source files have 't' as their extension, hence explicitly listing the extension is rather superfluous.

`USES_UNITS`

The names of the units depended upon by the `UNIT`. These units will appear in the `uses` clause of the unit file. If unspecified, `USES_UNITS` defaults to `tom`, i.e., the standard TOM library unit.

`LINK_UNITS`

The names of the units to be linked with the `UNIT` to produce the final result. For *bin* and *app* targets, this must include the `tom` unit, and every unit specified in the `USES_UNITS`. For *load* targets, this should include everything not already in the application into which loading is performed.

If unspecified, the `LINK_UNITS` are set equal to the `USES_UNITS` for *bin* and *app* targets, and set to `NONE` for *load* targets. `NONE` (case is important) means that the result will be linked against no other units.

Targets

The following standard makefile targets are defined by the TOM makefiles:

all

Build what is to be built. This usually is the default target.

clean

Remove targets and intermediate files.

gendoc

Extract the documentation from the TOM sources.

docclean

Remove the generated documentation.

More macros

The following macros can be useful -- they are documented and may thus be used:

`UNIT_PATH`

Specify directories (white-space separated) in which to search for TOM units. These directories are added to the default directory `$(tom_prefix)`. When looking for a unit *u*, for each directory *dir* in the path, the directory *dir*/*u* is checked for containing the unit *u*. The first match will be used.

For example, if your project `/home/me/src/myapp` uses the units tom, too, and the one called mylib in `/home/me/src/mylib`, the `UNIT_PATH` would be `..`, and `USES_UNITS` would include `mylib`. Note that the makefiles use the fact that a unit can reside in a subdirectory of the same name, in a directory somewhere along the `UNIT_PATH`. This is especially handy when using a lot of units, each residing in a subdirectory of a specific top-level directory.

`SUBPROJECTS`

The names of directories to be visited by make after the project in this directory is built, cleaned, etc. This macro usually is the only one used in a directory employing the `GNUmakefile.top`.

`C_SRC`

The names of any auxiliary C source files, without the extension. For example, the C unit has a file called `glue.c` which contains code that interfaces the TOM `Math` class with the C math library. Consequently, the GNUmakefile of the C unit specifies:

```
C_SRC= glue
```

Actually, the extension of the actual glue file does not matter, since for every word *mysrc* only the replacement *mysrc*.o is used in the makefiles. The makefiles however only contain the rule to create `$(GENDIR)/%.o` for every C source file `%.c`.

`GP_SRC`

Names of any sources to `gp`. As usual, these names exclude any extension. `GP_SRC` files have the extension `.tp`.

`EXTRA_OBJ`

Any extra object files to be linked with the unit.

Secondary GNUmakefiles

The TOM makefiles employ a file named `GNUmakefile.link` to list extra objects and libraries needed by the unit being built. The `GNUmakefile.link` is created by running the `$(TOMMAKE-FILES_DIR)/genlinkfile` shell script. `genlinkfile` tries to locate all unit files in the `$(LINK_UNITS)` along the $(UNIT_PATH), including the corresponding .la (libtool archive) files. When a unit is found but the archive isn't, as is the case with the installed TOM standard units, the name of a library directory is constructed in such a way that also the TOM standard units can be found in their installed place.

genlinkfile emits, for each unit found, lines that add to the following macros:

UNIT_OBJS

Directives (-l and -L) to have the linker link with the unit's library archive.

UNIT_DEPS

Files that the unit being built depends upon, suitable for use as a makefile dependency.

In addition, each unit can be accompanied by a GNUmakefile.unit; for every unit used, that makefile is included by the GNUmakefile.link. These GNUmakefile.unit files can add command line arguments to the linking phase of the unit being built:

UF_PRE_LIBS
UF_POST_LIBS

Linker arguments; UF_PRE_LIBS will precede all units' UF_POST_LIBS. This is, for example, used by a library that provides an abstraction of X11 and that uses AC_PATH_XTRA from autoconf AC_PATH_EXTRA sets X_PRE_LIBS, X_LIBS, and X_EXTRA_LIBS, and the library's GNUmakefile.unit.in will look like this:

```
UF_PRE_LIBS+= @X_PRE_LIBS@
UF_POST_LIBS+= @X_LIBS@ @X_EXTRA_LIBS@ -lX11
```

Environment Variables

Parts of the build process are controlled by Makefile macros, which can be usefully overridden by environment variables.

TESLA_FLAGS

Flags to pass to the compiler, tesla.

TIG_FLAGS

Flags to pass to the TOM interface generator, tig.

TUG_FLAGS

Flags to pass to the TOM unit generator, tug.

GPFLAGS

Flags to pass to the TOM parser generator, gp.

CFLAGS

Flags to pass to the compiler being used to compile the generated code. Typically, this will be gcc.

CPPFLAGS

Flags to pass to the compiler being used to compile the generated code. These will be passed before the command line options controlling includes..

LDFLAGS

Flags to pass to the linker when using either `GNUmakefile.app` or `GNUmakefile.bin` to build.

MFLAGS

Flags to pass to subsequent invocations of make.

Appendix B. GNU General Public License

Version 2, June 1991

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software--to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

GNU GENERAL PUBLIC LICENSE TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

1. 0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

2. 1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

3. 2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   a. a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

   b. b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

   c. c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

4. 3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

   a. a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   b. b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   c. c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

   The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

   If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

5. 4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is

void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

6. 5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

7. 6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

8. 7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

   If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

   It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

   This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

9. 8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

10. 9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

    Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

11. 10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

12. NO WARRANTY

    11. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

13. 12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

Appendix: How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
<one line to give the program's name and a brief idea of what it does.>
Copyright (C) 19yy  <name of author>

This program is free software; you can redistribute it
and/or modify it under the terms of the GNU General Public
License as published by the Free Software Foundation; either
version 2 of the License, or (at your option) any later
version.

This program is distributed in the hope that it will be
useful, but WITHOUT ANY WARRANTY; without even the implied
warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
See the GNU General Public License for more details.

You should have received a copy of the GNU General Public
License along with this program; if not, write to the Free
Software Foundation, Inc., 59 Temple Place - Suite 330,
Boston, MA 02111-1307, USA
```

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

Gnomovision version 69, Copyright (C) 19yy name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type ‘show w’.
This is free software, and you are welcome to redistribute it
under certain conditions; type ‘show c’ for details.

The hypothetical commands ‘show w’ and ‘show c’ should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than ‘show w’ and ‘show c’; they could even be mouse-clicks or menu items--whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright interest in the program
'Gnomovision' (which makes passes at compilers) written by James Hacker.

<signature of Ty Coon>, 1 April 1989
Ty Coon, President of Vice
```

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

Glossary

Class Compile Option

A `const` or a static class variable that is used as a compile-time option. All class compile options of a single unit are usually collected into a single class. Class compile options are not yet institutionalized -- they are a convention to-be.