GNU Smalltalk User's Guide

Version 3.2.5
24 November 2017

by Steven B. Byrne, Paolo Bonzini, Andy Valencia.

Introduction

GNU Smalltalk is an implementation that closely follows the Smalltalk-80 language as described in the book *Smalltalk-80: the Language and its Implementation* by Adele Goldberg and David Robson, which will hereinafter be referred to as *the Blue Book*.

The Smalltalk programming language is an object oriented programming language. This means, for one thing, that when programming you are thinking of not only the data that an object contains, but also of the operations available on that object. The object's data representation capabilities and the operations available on the object are "inseparable"; the set of things that you can do with an object is defined precisely by the set of operations, which Smalltalk calls *methods*, that are available for that object: each object belongs to a *class* (a datatype and the set of functions that operate on it) or, better, it is an *instance* of that class. You cannot even examine the contents of an object from the outside—to an outsider, the object is a black box that has some state and some operations available, but that's all you know: when you want to perform an operation on an object, you can only send it a *message*, and the object picks up the method that corresponds to that message.

In the Smalltalk language, everything is an object. This includes not only numbers and all data structures, but even classes, methods, pieces of code within a method (*blocks* or *closures*), stack frames (*contexts*), etc. Even `if` and `while` structures are implemented as methods sent to particular objects.

Unlike other Smalltalks (including Smalltalk-80), GNU Smalltalk emphasizes Smalltalk's rapid prototyping features rather than the graphical and easy-to-use nature of the programming environment (did you know that the first GUIs ever ran under Smalltalk?). The availability of a large body of system classes, once you master them, makes it pretty easy to write complex programs which are usually a task for the so called *scripting languages*. Therefore, even though we have a GUI environment based on GTK (see Section 3.1 [GTK and VisualGST], page 34), the goal of the GNU Smalltalk project is currently to produce a complete system to be used to write your scripts in a clear, aesthetically pleasing, and philosophically appealing programming language.

An example of what can be obtained with Smalltalk in this novel way can be found in Section "Class reference" in *the GNU Smalltalk Library Reference*. That part of the manual is entirely generated by a Smalltalk program, starting from the source code for the class libraries distributed together with the system.

1 Using GNU Smalltalk

1.1 Command line arguments

The GNU Smalltalk virtual machine may be invoked via the following command:

```
gst [ flags ... ] [ file ... ]
```

When you invoke GNU Smalltalk, it will ensure that the binary image file (called `gst.im`) is up to date; if not, it will build a new one as described in Section 1.2.2 [Loading an image or creating a new one], page 8. Your first invocation should look something like this:

```
"Global garbage collection... done"
GNU Smalltalk ready

st>
```

If you specify one or more *file*s, they will be read and executed in order, and Smalltalk will exit when end of file is reached. If you don't specify *file*, GNU Smalltalk reads standard input, issuing a '`st>`' prompt if the standard input is a terminal. You may specify `-` for the name of a file to invoke an explicit read from standard input.

To exit while at the '`st>`' prompt, use *`Ctrl-d`*, or type *`ObjectMemory quit`* followed by `RET`. Use *`ObjectMemory snapshot`* first to save a new image that you can reload later, if you wish.

As is standard for GNU-style options, specifying `--` stops the interpretation of options so that every argument that follows is considered a file name even if it begins with a '`-`'.

You can specify both short and long flags; for example, `--version` is exactly the same as `-v`, but is easier to remember. Short flags may be specified one at a time, or in a group. A short flag or a group of short flags always starts off with a single dash to indicate that what follows is a flag or set of flags instead of a file name; a long flag starts off with two consecutive dashes, without spaces between them.

In the current implementation the flags can be intermixed with file names, but their effect is as if they were all specified first. The various flags are interpreted as follows:

`-a`
`--smalltalk-args`
: Treat all options afterward as arguments to be given to Smalltalk code retrievable with `Smalltalk arguments`, ignoring them as arguments to GNU Smalltalk itself.

  Examples:

command line	Options seen by GNU Smalltalk	`Smalltalk arguments`
(empty)	(none)	`#()`
`-Via foo bar`	`-Vi`	`#('foo' 'bar')`
`-Vai test`	`-Vi`	`#('test')`
`-Vaq`	`-Vq`	`#()`
`--verbose -aq -c`	`--verbose -q`	`#('-c')`

`-c`
`--core-dump`
When a fatal signal occurs, produce a core dump before terminating. Without this option, only a backtrace is provided.

`-D`
`--declaration-trace`
Print the class name, the method name, and the byte codes that the compiler generates as it compiles methods. Only applies to files that are named explicitly on the command line, unless the flag is given multiple times on the command line.

`-E`
`--execution-trace`
Print the byte codes being executed as the interpreter operates. Only works for statements explicitly issued by the user (either interactively or from files given on the command line), unless the flag is given multiple times on the command line.

`--kernel-directory`
Specify the directory from which the kernel source files will be loaded. This is used mostly while compiling GNU Smalltalk itself. Smalltalk code can retrieve this information with `Directory kernel`.

`--no-user-files`
Don't load any files from `~/.st/` (see Section 1.2.2 [Loading an image or creating a new one], page 8).[1] This is used mostly while compiling GNU Smalltalk itself, to ensure that the installed image is built only from files in the source tree.

`-K` *file*
`--kernel-file` *file*
Load *file* in the usual way, but look for it relative to the kernel directory's parent directory, which is usually `/usr/local/share/smalltalk/`. See `--kernel-dir` above.

`-f`
`--file` The following two command lines are equivalent:

```
gst -f file args...
gst -q file -a args...
```

This is meant to be used in the so called "sharp-bang" sequence at the beginning of a file, as in

```
#! /usr/bin/gst -f

... Smalltalk source code ...
```

[1] The directory would be called `_st/` under MS-DOS. Under OSes that don't use home directories, it would be looked for in the current directory.

GNU Smalltalk treats the first line as a comment, and the `-f` option ensures that the arguments are passed properly to the script. Use this instead to avoid hard-coding the path to `gst`:[2]

```
#! /bin/sh
"exec" "gst" "-f" "$0" "$@"

... Smalltalk source code ...
```

`-g`
`--no-gc-messages`
: Suppress garbage collection messages.

`-h`
`--help`
: Print out a brief summary of the command line syntax of GNU Smalltalk, including the definitions of all of the option flags, and then exit.

`-i`
`--rebuild-image`
: Always build and save a new image file; see Section 1.2.2 [Loading an image or creating a new one], page 8.

`--maybe-rebuild-image`
: Perform the image checks and rebuild as described in Section 1.2.2 [Loading an image or creating a new one], page 8. This is the default when `-I` is not given.

`-I` *file*
`--image-file` *file*
: Use the image file named *file* as the image file to load instead of the default location, and set *file*'s directory part as the image path. This option completely bypasses checking the file dates on the kernel files; use `--maybe-rebuild-image` to restore the usual behavior, writing the newly built image to *file* if needed.

`-q`
`--quiet`
`--silent`
: Suppress the printing of answered values from top-level expressions while GNU Smalltalk runs.

`-r`
`--regression-test`
: This is used by the regression testing system and is probably not of interest to the general user. It controls printing of certain information.

`-S`
`--snapshot`
: Save the image after loading files from the command line. Of course this "snapshot" is not saved if you include - (stdin) on the command line and exit by typing *`Ctrl-c`*.

[2] The words in the shell command `exec` are all quoted, so GNU Smalltalk parses them as five separate comments.

`-v`
`--version`
: Print out the GNU Smalltalk version number, then exit.

`-V`
`--verbose`
: Print various diagnostic messages while executing (the name of each file as it's loaded, plus messages about the beginning of execution or how many byte codes were executed).

1.2 Startup sequence

Caveat: *The startup sequence is pretty complicated. If you are not interested in its customization, you can skip the first two sections below. These two sections also don't apply when using the command-line option* `-I`*, unless also using* `--maybe-rebuild-image`*.*

You can abort GNU Smalltalk at any time during this procedure with *`Ctrl-c`*.

1.2.1 Picking an image path and a kernel path

When GNU Smalltalk is invoked, it first chooses two paths, the "image path" and the "kernel path". The image path is set by considering these paths in succession:

- the directory part of the `--image-file` option if it is given;
- the value of the `SMALLTALK_IMAGE` environment variable if it is defined and readable; this step will disappear in a future release;
- the path compiled in the binary (usually, under Unix systems, `/usr/local/var/lib/smalltalk` or a similar path under `/var`) if it exists and it is readable;
- the current directory. The current directory is also used if the image has to be rebuilt but you cannot write to a directory chosen according to the previous criteria.

The "kernel path" is the directory in which to look for Smalltalk code compiled into the base image. The possibilities in this case are:

- the argument to the `--kernel-dir` option if it is given;
- the value of the `SMALLTALK_KERNEL` environment variable if it is defined and readable; this step will disappear in a future release;
- the path compiled in the binary (usually, under Unix systems, `/usr/local/share/smalltalk/kernel` or a similar data file path) if it exists and it is readable;
- a subdirectory named `kernel` of the image path.

1.2.2 Loading an image or creating a new one

GNU Smalltalk can load images created on any system with the same pointer size as its host system by approximately the same version of GNU Smalltalk, even if they have different endianness. For example, images created on 32-bit PowerPC can be loaded with a 32-bit x86 `gst` VM, provided that the GNU Smalltalk versions are similar enough. Such images are called *compatible images*. It cannot load images created on systems with different pointer sizes; for example, our x86 `gst` cannot load an image created on x86-64.

Unless the `-i` flag is used, GNU Smalltalk first tries to load the file named by `--image-file`, defaulting to `gst.im` in the image path. If this is found, GNU Smalltalk ensures the

image is "not stale", meaning its write date is newer than the write dates of all of the kernel method definition files. It also ensures that the image is "compatible", as described above. If both tests pass, GNU Smalltalk loads the image and continues with Section 1.2.3 [After the image is created or restored], page 9.

If that fails, a new image has to be created. The image path may now be changed to the current directory if the previous choice is not writeable.

To build an image, GNU Smalltalk loads the set of files that make up the kernel, one at a time. The list can be found in `libgst/lib.c`, in the `standard_files` variable. You can override kernel files by placing your own copies in `~/.st/kernel/`.[3] For example, if you create a file `~/.st/kernel/Builtins.st`, it will be loaded instead of the `Builtins.st` in the kernel path.

To aid with image customization and local bug fixes, GNU Smalltalk loads two more files (if present) before saving the image. The first is `site-pre.st`, found in the parent directory of the kernel directory. Unless users at a site change the kernel directory when running `gst`, `/usr/local/share/smalltalk/site-pre.st` provides a convenient place for site-wide customization. The second is `~/.st/pre.st`, which can be different for each user's home directory.[4].

Before the next steps, GNU Smalltalk takes a snapshot of the new memory image, saving it over the old image file if it can, or in the current directory otherwise.

1.2.3 After the image is created or restored

Next, GNU Smalltalk sends the `returnFromSnapshot` event to the dependents of the special class `ObjectMemory` (see Section 2.8 [Memory access], page 23). Afterwards, it loads `~/.st/init.st` if available.[5]

You can remember the difference between `pre.st` and `init.st` by remembering that `pre.st` is the *pre*-snapshot file and `init.st` is the post-image-load *init*ialization file.

Finally, GNU Smalltalk loads files listed on the command line, or prompts for input at the terminal, as described in Section 1.1 [Command line arguments], page 5.

1.3 Syntax of GNU Smalltalk

The language that GNU Smalltalk accepts is basically the same that other Smalltalk environment accept and the same syntax used in the *Blue Book*, also known as *Smalltalk-80: The Language and Its Implementation*. The return operator, which is represented in the Blue Book as an up-arrow, is mapped to the ASCII caret symbol `^`; the assignment operator (left-arrow) is usually represented as `:=`[6].

---

[3] The directory is called `_st/kernel` under MS-DOS. Under OSes that don't use home directories, it is looked for in the current directory.

[4] The file is looked up as `_st/pre.st` under MS-DOS and again, under OSes that don't use home directories it is looked for as `pre.st` in the current directory.

[5] The same considerations made above hold here too. The file is called `_st/init.st` under MS-DOS, and is looked for in the current directory under OSes that don't use home directories.

[6] It also bears mentioning that there are two assignment operators: `_` and `:=`. Both are usable interchangeably, provided that they are surrounded by spaces. The GNU Smalltalk kernel code uses the `:=` form exclusively, but `_` is supported a) for compatibility with previous versions of GNU Smalltalk b) because this is the correct mapping between the assignment operator mentioned in the Blue Book and the current

Actually, the grammar of GNU Smalltalk is slightly different from the grammar of other Smalltalk environments in order to simplify interaction with the system in a command-line environment as well as in full-screen editors.

Statements are executed one by one; multiple statements are separated by a period. At end-of-line, if a valid statement is complete, a period is implicit. For example,

```
8r300. 16rFFFF
```

prints out the decimal value of octal `300` and hex `FFFF`, each followed by a newline.

Multiple statements share the same local variables, which are automatically declared. To delete the local variables, terminate a statement with `!` rather than `.` or newline. Here,

```
a := 42
a!
a
```

the first two `a`s are printed as `42`, but the third one is uninitialized and thus printed as `nil`.

In order to evaluate multiple statements in a single block, wrap them into an *eval block* as follows:

```
Eval [
    a := 42.  a printString
]
```

This won't print the intermediate result (the integer 42), only the final result (the string `'42'`).

```
ObjectMemory quit
```

exits from the system. You can also type a *C-d* to exit from Smalltalk if it's reading statements from standard input.

GNU Smalltalk provides three extensions to the language that make it simpler to write complete programs in an editor. However, it is also compatible with the *file out* syntax as shown in the *Green Book* (also known as *Smalltalk-80: Bits of History, Words of Advice* by Glenn Krasner).

A new class is created using this syntax:

superclass-name `subclass:` *new-class-name* `[`
`|` *instance variables* `|`
pragmas
message-pattern-1 `[` *statements* `]`
message-pattern-2 `[` *statements* `]`
...
class-variable-1 `:=` *expression*`.`
class-variable-2 `:=` *expression*`.`
...
`]`

In short:

- Instance variables are defined with the same syntax as method temporary variables.

ASCII definition. In the ancient days (like the middle 70's), the ASCII underscore character was also printed as a back-arrow, and many terminals would display it that way, thus its current usage. Anyway, using _ may lead to portability problems.

- Unlike other Smalltalks, method statements are inside brackets.
- Class variables are defined the same as variable assignments.
- Pragmas define class comment, class category, imported namespaces, and the shape of indexed instance variables.

```
<comment: 'Class comment'>
<category: 'Examples-Intriguing'>
<import: SystemExceptions>
<shape: #pointer>
```

A similar syntax is used to define new methods in an existing class.

class-expression `extend [`
`...`
`]`

The *class-expression* is an expression that evaluates to a class object, which is typically just the name of a class, although it can be the name of a class followed by the word `class`, which causes the method definitions that follow to apply to the named class itself, rather than to its instances.

```
Number extend [
    radiusToArea [
        ^self squared * Float pi
    ]
    radiusToCircumference [
        ^self * 2 * Float pi
    ]
]
```

A complete treatment of the Smalltalk syntax and of the class library can be found in the included tutorial and class reference (see Section "Class Reference" in *the* GNU *Smalltalk Library Reference*).

More information on the implementation of the language can be found in the *Blue Book*; the relevant parts are available, scanned, at `http://stephane.ducasse.free.fr/FreeBooks/BlueBook/Bluebook.pdf`.

1.4 Running the test suite

GNU Smalltalk comes with a set of files that provides a simple regression test suite.

To run the test suite, you should be connected to the top-level Smalltalk directory. Type

```
make check
```

You should see the names of the test suite files as they are processed, but that's it. Any other output indicates some problem.

1.5 Licensing of GNU Smalltalk

Different parts of GNU Smalltalk comes under two licenses: the virtual machine and the development environment (compiler and browser) come under the GNU General Public License, while the system class libraries come under the Lesser General Public License.

1.5.1 Complying with the GNU GPL

The GPL licensing of the virtual machine means that all derivatives of the virtual machine must be put under the same license. In other words, it is strictly forbidden to distribute programs that include the GNU Smalltalk virtual machine under a license that is not the GPL. This also includes any bindings to external libraries. For example, the bindings to Gtk+ are released under the GPL.

In principle, the GPL would not extend to Smalltalk programs, since these are merely input data for the virtual machine. On the other hand, using bindings that are under the GPL via dynamic linking would constitute combining two parts (the Smalltalk program and the bindings) into one program. Therefore, we added a special exception to the GPL in order to avoid gray areas that could adversely hit both the project and its users:

> In addition, as a special exception, the Free Software Foundation give you permission to combine GNU Smalltalk with free software programs or libraries that are released under the GNU LGPL and with independent programs running under the GNU Smalltalk virtual machine.
>
> You may copy and distribute such a system following the terms of the GNU GPL for GNU Smalltalk and the licenses of the other code concerned, provided that you include the source code of that other code when and as the GNU GPL requires distribution of source code.
>
> Note that people who make modified versions of GNU Smalltalk are not obligated to grant this special exception for their modified versions; it is their choice whether to do so. The GNU General Public License gives permission to release a modified version without this exception; this exception also makes it possible to release a modified version which carries forward this exception.

1.5.2 Complying with the GNU LGPL

Smalltalk programs that run under GNU Smalltalk are linked with the system classes in GNU Smalltalk class library. Therefore, they must respect the terms of the Lesser General Public License[7].

The interpretation of this license for architectures different from that of the C language is often difficult; the accepted one for Smalltalk is as follows. The image file can be considered as an object file, falling under Subsection 6a of the license, as long as it allows a user to load an image, upgrade the library or otherwise apply modifications to it, and save a modified image: this is most conveniently obtained by allowing the user to use the read-eval-print loop that is embedded in the GNU Smalltalk virtual machine.

In other words, provided that you leave access to the loop in a documented way, or that you provide a way to file in arbitrary files in an image and save the result to a new image, you are obeying Subsection 6a of the Lesser General Public License, which is reported here:

> a) Accompany the work with the complete corresponding machine-readable source code for the Library including whatever changes were used in the work (which must be distributed under Sections 1 and 2 above); and, if the work is an executable linked with the Library, with the complete machine-readable "work that uses the Library", as object code and/or source code, so that the

[7] Of course, they may be more constrained by usage of GPL class libraries.

user can modify the Library and then relink to produce a modified executable containing the modified Library. (It is understood that the user who changes the contents of definitions files in the Library will not necessarily be able to recompile the application to use the modified definitions.)

In the future, alternative mechanisms similar to shared libraries may be provided, so that it is possible to comply with the GNU LGPL in other ways.

2 Features of GNU Smalltalk

In this section, the features which are specific to GNU Smalltalk are described. These features include support for calling C functions from within Smalltalk, accessing environment variables, and controlling various aspects of compilation and execution monitoring.

Note that, in general, GNU Smalltalk is much more powerful than the original Smalltalk-80, as it contains a lot of methods that are common in today's Smalltalk implementation and are present in the ANSI Standard for Smalltalk, but were absent in the Blue Book. Examples include Collection's `allSatisfy:` and `anySatisfy:` methods and many methods in SystemDictionary (the Smalltalk dictionary's class).

2.1 Extended streams

The basic image in GNU Smalltalk includes powerful extensions to the *Stream* hierarchy found in ANSI Smalltalk (and Smalltalk-80). In particular:

- Read streams support all the iteration protocols available for collections. In some cases (like `fold:`, `detect:`, `inject:into:`) these are completely identical. For messages that return a new stream, such as `select:` and `collect:`, the blocks are evaluated lazily, as elements are requested from the stream using `next`.
- Read streams can be concatenated using `,` like SequenceableCollections.
- *Generators* are supported as a quick way to create a Stream. A generator is a kind of pluggable stream, in that a user-supplied blocks defines which values are in a stream.

  For example, here is an empty generator and two infinite generators:

  ```
  "Returns an empty stream"
  Generator on: [ :gen | ]

  "Return an infinite stream of 1's"
  Generator on: [ :gen | [ gen yield: 1 ] repeat ]

  "Return an infinite stream of integers counting up from 1"
  Generator inject: 1 into: [ :value | value + 1 ]
  ```

  The block is put "on hold" and starts executing as soon as `#next` or `#atEnd` are sent to the generator. When the block sends `#yield:` to the generator, it is again put on hold and the argument becomes the next object in the stream.

  Generators use *continuations*, but they shield the users from their complexity by presenting the same simple interface as streams.

2.2 Regular expression matching

Regular expressions, or "regexes", are a sophisticated way to efficiently match patterns of text. If you are unfamiliar with regular expressions in general, see Section "20.5 Syntax of Regular Expressions" in *GNU Emacs Manual*, for a guide for those who have never used regular expressions.

GNU Smalltalk supports regular expressions in the core image with methods on `String`.

The GNU Smalltalk regular expression library is derived from GNU libc, with modifications made originally for Ruby to support Perl-like syntax. It will always use its included

library, and never the ones installed on your system; this may change in the future in backwards-compatible ways. Regular expressions are currently 8-bit clean, meaning they can work with any ordinary String, but do not support full Unicode, even when package `I18N` is loaded.

Broadly speaking, these regexes support Perl 5 syntax; register groups '`()`' and repetition '`{}`' must not be given with backslashes, and their counterpart literal characters should. For example, '`\{{1,3}`' matches '`{`', '`{{`', '`{{{`'; correspondingly, '`(a)(\()`' matches '`a(`', with '`a`' and '`(`' as the first and second register groups respectively. GNU Smalltalk also supports the regex modifiers '`imsx`', as in Perl. You can't put regex modifiers like '`im`' after Smalltalk strings to specify them, because they aren't part of Smalltalk syntax. Instead, use the inline modifier syntax. For example, '`(?is:abc.)`' is equivalent to '`[Aa][Bb][Cc](?:.|\n)`'.

In most cases, you should specify regular expressions as ordinary strings. GNU Smalltalk always caches compiled regexes, and uses a special high-efficiency caching when looking up literal strings (i.e. most regexes), to hide the compiled `Regex` objects from most code. For special cases where this caching is not good enough, simply send `#asRegex` to a string to retrieved a compiled form, which works in all places in the public API where you would specify a regex string. You should always rely on the cache until you have demonstrated that using Regex objects makes a noticeable performance difference in your code.

Smalltalk strings only have one escape, the '' given by '''', so backslashes used in regular expression strings will be understood as backslashes, and a literal backslash can be given directly with '`\\`'[1].

The methods on the compiled Regex object are private to this interface. As a public interface, GNU Smalltalk provides methods on String, in the category '`regex`'. There are several methods for matching, replacing, pattern expansion, iterating over matches, and other useful things.

The fundamental operator is `#searchRegex:`, usually written as `#=~`, reminiscent of Perl syntax. This method will always return a `RegexResults`, which you can query for whether the regex matched, the location Interval and contents of the match and any register groups as a collection, and other features. For example, here is a simple configuration file line parser:

```
| file config |
config := LookupTable new.
file := (File name: 'myapp.conf') readStream.
file linesDo: [:line |
    (line =~ '(\w+)\s*=\s*((?: ?\w+)+)') ifMatched: [:match |
        config at: (match at: 1) put: (match at: 2)]].
file close.
config printNl.
```

As with Perl, `=~` will scan the entire string and answer the leftmost match if any is to be found, consuming as many characters as possible from that position. You can anchor the search with variant messages like `#matchRegex:`, or of course `^` and `$` with their usual semantics if you prefer.

[1] Whereas it must be given as '`\\\\`' in a literal Emacs Lisp string, for example.

You shouldn't modify the string while you want a particular RegexResults object matched on it to remain valid, because changes to the matched text may propagate to the RegexResults object.

Analogously to the Perl `s` operator, GNU Smalltalk provides `#replacingRegex:with:`. Unlike Perl, GNU Smalltalk employs the pattern expansion syntax of the `#%` message here. For example, `'The ratio is 16/9.' replacingRegex: '(\d+)/(\d+)' with: '$%1\over%2$'` answers `'The ratio is $16\over9$.'`. In place of the `g` modifier, use the `#replacingAllRegex:with:` message instead.

One other interesting String message is `#onOccurrencesOfRegex:do:`, which invokes its second argument, a block, on every successful match found in the receiver. Internally, every search will start at the end of the previous successful match. For example, this will print all the words in a stream:

```
stream contents onOccurrencesOfRegex: '\w+'
                do: [:each | each match printNl]
```

2.3 Namespaces

[This section (and the implementation of namespaces in GNU Smalltalk) is based on the paper Structured Symbolic Name Spaces in Smalltalk, by Augustin Mrazik.]

2.3.1 Introduction

The Smalltalk-80 programming environment, upon which GNU Smalltalk is historically based, supports symbolic identification of objects in one global namespace—in the `Smalltalk` system dictionary. This means that each global variable in the system has its unique name which is used for symbolic identification of the particular object in the source code (e.g. in expressions or methods). The most important of these global variables are classes defining the behavior of objects.

In development dealing with modelling of real systems, *polymorphic symbolic identification* is often needed. By this, we mean that it should be possible to use the same name for different classes or other global variables. Selection of the proper variable binding should be context-specific. By way of illustration, let us consider class `Statement` as an example which would mean totally different things in different domains:

GNU Smalltalk or other programming language
: An expression in the top level of a code body, possibly with special syntax available such as assignment or branching.

Bank
: A customer's trace report of recent transactions.

AI, logical derivation
: An assertion of a truth within a logical system.

This issue becomes inevitable if we start to work persistently, using `ObjectMemory snapshot` to save after each session for later resumption. For example, you might have the class `Statement` already in your image with the "Bank" meaning above (e.g. in the live bank support systems we all run in our images) and you might decide to start developing YAC [Yet Another C]. Upon starting to write parse nodes for the compiler, you would find that `#Statement` is boundk in the banking package. You could replace it with your parse

node class, and the bank's `Statement` could remain in the system as an unbound class with full functionality; however, it could not be accessed anymore at the symbolic level in the source code. Whether this would be a problem or not would depend on whether any of the bank's code refers to the class `Statement`, and when these references occur.

Objects which have to be identified in source code by their names are included in `Smalltalk`, the sole instance of `SystemDictionary`. Such objects may be identified simply by writing their names as you would any variable names. The code is compiled in the default environment, and if the variable is found in `Smalltalk`, without being shadowed by a class pool or local variables, its value is retrieved and used as the value of the expression. In this way `Smalltalk` represents the sole symbolic namespace. In the following text the symbolic namespace, as a concept, will be called simply *environment* to make the text more clear.

2.3.2 Concepts

To support polymorphic symbolical identification several environments will be needed. The same name may exist concurrently in several environments as a key, pointing to diverse objects in each.

Symbolic navigation between these environments is needed. Before approaching the problem of the syntax and semantics to be implemented, we have to decide on structural relations to be established between environments.

Since the environment must first be symbolically identified to direct access to its global variables, it must first itself be a global variable in another environment. `Smalltalk` is a great choice for the root environment, from which selection of other environments and their variables begins. From `Smalltalk` some of the existing sub-environments may be seen; from these other sub-environments may be seen, etc. This means that environments represent nodes in a graph where symbolic selections from one environment to another one represent branches.

The symbolic identification should be unambiguous, although it will be polymorphic. This is why we should avoid cycles in the environment graph. Cycles in the graph could cause also other problems in the implementation, e.g. inability to use trivially recursive algorithms. Thus, in general, the environments must build a directed acyclic graph; GNU Smalltalk currently limits this to an n-ary tree, with the extra feature that environments can be used as pool dictionaries.

Let us call the partial ordering relation which occurs between environments *inheritance*. Sub-environments inherit from their super-environments. The feature of inheritance in the meaning of object-orientation is associated with this relation: all associations of the super-environment are valid also in its sub-environments, unless they are locally redefined in the sub-environment.

A super-environment includes all its sub-enviroments as `Associations` under their names. The sub-environment includes its super-environment under the symbol `#Super`. Most environments inherit from `Smalltalk`, the standard root environment, but they are not required to do so; this is similar to how most classes derive from `Object`, yet one can derive a class directly from `nil`. Since they all inherit `Smalltalk`'s global variables, it is not necessary to define `Smalltalk` as pointing to `Smalltalk`'s `Smalltalk` in each environment.

The inheritance links to the super-environments are used in the lookup for a potentially inherited global variable. This includes lookups by a compiler searching for a variable binding and lookups via methods such as `#at:` and `#includesKey:`.

2.3.3 Syntax

Global objects of an environment, be they local or inherited, may be referenced by their symbol variable names used in the source code, e.g.

```
John goHome
```

if the `#John -> aMan` association exists in the particular environment or one of its super-environments, all along the way to the root environment.

If an object must be referenced from another environment (i.e. which is not one of its sub-environments) it has to be referenced either *relatively* to the position of the current environment, using the `Super` symbol, or *absolutely*, using the "full pathname" of the object, navigating from the tree root (usually `Smalltalk`) through the tree of sub-environments.

For the identification of global objects in another environment, we use a "pathname" of symbols. The symbols are separated by periods; the "look" to appear is that of

```
Smalltalk.Tasks.MyTask
```

and of

```
Super.Super.Peter.
```

As is custom in Smalltalk, we are reminded by capitalization that we are accessing global objects. Another syntax returns the *variable binding*, the `Association` for a particular global. The first example above is equivalently:

```
#{Smalltalk.Tasks.MyTask} value
```

The latter syntax, a *variable binding*, is also valid inside literal arrays.

2.3.4 Implementation

A superclass of `SystemDictionary` called `RootNamespace` is defined, and many of the features of the Smalltalk-80 `SystemDictionary` will be hosted by that class. `Namespace` and `RootNamespace` are in turn subclasses of `AbstractNamespace`.

To handle inheritance, the following methods have to be defined or redefined in Namespace (*not* in RootNamespace):

Accessors like `#at:ifAbsent:` and `#includesKey:`
: Inheritance must be implemented. When `Namespace`, trying to read a variable, finds an association in its own dictionary or a super-environment dictionary, it uses that; for `Dictionary`'s writes and when a new association must be created, `Namespace` creates it in its own dictionary. There are special methods like `#set:to:` for cases in which you want to modify a binding in a super-environment if that is the relevant variable's binding.

Enumerators like `#do:` and `#keys`
: This should return **all** the objects in the namespace, including those which are inherited.

Hierarchy access

`AbstractNamespace` will also implement a new set of methods that allow one to navigate through the namespace hierarchy; these parallel those found in `Behavior` for the class hierarchy.

The most important task of the `Namespace` class is to provide organization for the most important global objects in the Smalltalk system—for the classes. This importance becomes even more crucial in a structure of multiple environments intended to change the semantics of code compiled for those classes.

In Smalltalk the classes have the instance variable `name` which holds the name of the class. Each *defined class* is included in `Smalltalk`, or another environment, under this name. In a framework with several environments the class should know the environment in which it has been created and compiled. This is a new property of `Class` which must be defined and properly used in relevant methods. In the mother environment the class shall be included under its name.

Any class, as with any other object, may be included concurrently in several environments, even under different symbols in the same or in diverse environments. We can consider these "alias names" of the particular class or other value. A class may be referenced under the other names or in other environments than its mother environment, e.g. for the purpose of instance creation or messages to the class, but it should not compile code in these environments, even if this compilation is requested from another environment. If the syntax is not correct in the mother environment, a compilation error occurs. This follows from the existence of class "mother environments", as a class is responsible for compiling its own methods.

An important issue is also the name of the class answered by the class for the purpose of its identification in diverse tools (e.g. in a browser). This must be changed to reflect the environment in which it is shown, i.e. the method '`nameIn: environment`' must be implemented and used in proper places.

Other changes must be made to the Smalltalk system to achieve the full functionality of structured environments. In particular, changes have to be made to the behavior classes, the user interface, the compiler, and a few classes supporting persistance. One small detail of note is that evaluation in the REPL or '`Workspace`', implemented by compiling methods on `UndefinedObject`, make more sense if `UndefinedObject`'s environment is the "current environment" as reachable by `Namespace current`, even though its mother environment by any other sensibility is `Smalltalk`.

2.3.5 Using namespaces

Using namespaces is often merely a matter of adding a '`namespace`' option to the GNU Smalltalk XML package description used by `PackageLoader`, or wrapping your code like this:

```
Namespace current: NewNS [
    ...
]
```

Namespaces can be imported into classes like this:

```
Stream subclass: EncodedStream [
    <import: Encoders>
```

```
]
```

Alternatively, paths to classes (and other objects) in the namespaces will have to be specified completely. Importing a namespace into a class is similar to C++'s `using namespace` declaration within the class proper's definition.

Finally, be careful when working with fundamental system classes. Although you can use code like

```
Namespace current: NewNS [
    Smalltalk.Set subclass: Set [
        <category: 'My application-Extensions'>
        ...
    ]
]
```

this approach won't work when applied to core classes. For example, you might be successful with a `Set` or `WriteStream` object, but subclassing `SmallInteger` this way can bite you in strange ways: integer literals will still belong to the `Smalltalk` dictionary's version of the class (this holds for `Array`s, `String`s, etc. too), primitive operations will still answer standard Smalltalk `SmallIntegers`, and so on. Similarly, word-shaped will recognize 32-bit `Smalltalk.LargeInteger` objects, but not `LargeInteger`s belonging to your own namespace.

Unfortunately, this problem is not easy to solve since Smalltalk has to know the OOPs of determinate class objects for speed—it would not be feasible to lookup the environment to which sender of a message belongs every time the `+` message was sent to an Integer.

So, GNU Smalltalk namespaces cannot yet solve 100% of the problem of clashes between extensions to a class—for that you'll still have to rely on prefixes to method names. But they *do* solve the problem of clashes between class names, or between class names and pool dictionary names.

Namespaces are unrelated from packages; loading a package does not import the corresponding namespace.

2.4 Disk file-IO primitive messages

Four classes (`FileDescriptor`, `FileStream`, `File`, `Directory`) allow you to create files and access the file system in a fully object-oriented way.

`FileDescriptor` and `FileStream` are much more powerful than the corresponding C language facilities (the difference between the two is that, like the C `stdio` library, `FileStream` does buffering). For one thing, they allow you to write raw binary data in a portable endian-neutral format. But, more importantly, these classes transparently implement virtual filesystems and asynchronous I/O.

Asynchronous I/O means that an input/output operation blocks the Smalltalk Process that is doing it, but not the others, which makes them very useful in the context of network programming. Virtual file systems mean that these objects can transparently extract files from archives such as `tar` and `gzip` files, through a mechanism that can be extended through either shell scripting or Smalltalk programming. For more information on these classes, look in the class reference, under the `VFS` namespace. URLs may be used as file

names; though, unless you have loaded the `NetClients` package (see Section 3.8 [Network support], page 44), only `file` URLs will be accepted.

In addition, the three files, `stdin`, `stdout`, and `stderr` are declared as global instances of `FileStream` that are bound to the proper values as passed to the C virtual machine. They can be accessed as either `stdout` and `FileStream stdout`—the former is easier to type, but the latter can be clearer.

Finally, `Object` defines four other methods: `print` and `printNl`, `store` and `storeNl`. These do a `printOn:` or `storeOn:` to the "Transcript" object; this object, which is the sole instance of class `TextCollector`, normally delegates write operations to `stdout`. If you load the VisualGST GUI, instead, the Transcript Window will be attached to the Transcript object (see Section 3.1 [GTK and VisualGST], page 34).

The `fileIn:` message sent to the FileStream class, with a file name as a string argument, will cause that file to be loaded into Smalltalk.

For example,

```
FileStream fileIn: 'foo.st' !
```

will cause `foo.st` to be loaded into GNU Smalltalk.

2.5 The GNU Smalltalk ObjectDumper

Another GNU Smalltalk-specific class, the `ObjectDumper` class, allows you to dump objects in a portable, endian-neutral, binary format. Note that you can use the `ObjectDumper` on ByteArrays too, thanks to another GNU Smalltalk-specific class, `ByteStream`, which allows you to treat ByteArrays the same way you would treat disk files.

For more information on the usage of the `ObjectDumper`, look in the class reference.

2.6 Dynamic loading

The `DLD` class enhances the C callout mechanism to automatically look for unresolved functions in a series of program-specified libraries. To add a library to the list, evaluate code like the following:

```
DLD addLibrary: 'libc'
```

The extension (`.so`, `.sl`, `.a`, `.dll` depending on your operating system) will be added automatically. You are advised not to specify it for portability reasons.

You will then be able to use the standard C call-out mechanisms to define all the functions in the C run-time library. Note that this is a potential security problem (especially if your program is SUID root under Unix), so you might want to disable dynamic loading when using GNU Smalltalk as an extension language. To disable dynamic loading, configure GNU Smalltalk passing the `--disable-dld` switch.

Note that a `DLD` class will be present even if dynamic loading is disabled (either because your system is not supported, or by the `--disable-dld` configure switch) but any attempt to perform dynamic linking will result in an error.

2.7 Automatic documentation generator

GNU Smalltalk includes an automatic documentation generator invoked via the `gst-doc` command. The code is actually part of the `ClassPublisher` package, and `gst-doc` takes care of reading the code to be documented and firing a `ClassPublisher`.

Currently, `gst-doc` can only generate output in Texinfo format, though this will change in future releases.

`gst-doc` can document code that is already in the image, or it can load external files and packages. Note that the latter approach will not work for files and packages that programmatically create code or file in other files/packages.

`gst-doc` is invoked as follows:

```
gst-doc [ flag ... ] class ...
```

The following options are supported:

`-p` *package*
`--package=`*package*
: Produce documentation for the classes inside the *package* package.

`-f` *file*
`--file=`*file*
: Produce documentation for the classes inside the *file* file.

`-I`
`--image-file`
: Produce documentation for the code that is already in the given image.

`-o`
`--output=`*file*
: Emit documentation in the named file.

class is either a class name, or a namespace name followed by `.*`. Documentation will be written for classes that are specified in the command line. *class* can be omitted if a `-f` or `-p` option is given. In this case, documentation will be written for all the classes in the package.

2.8 Memory accessing methods

GNU Smalltalk provides methods to query its own internal data structures. You may determine the real memory address of an object or the real memory address of the OOP table that points to a given object, by using messages to the `Memory` class, described below.

`asOop` [Method on `Object`]
: Returns the index of the OOP for anObject. This index is immume from garbage collection and is the same value used by default as an hash value for anObject (it is returned by Object's implementation of `hash` and `identityHash`).

`asObject` [Method on `Integer`]
: Converts the given OOP *index* (not address) back to an object. Fails if no object is associated to the given index.

`asObjectNoFail` [Method on `Integer`]
: Converts the given OOP *index* (not address) back to an object. Returns nil if no object is associated to the given index.

Other methods in ByteArray and Memory allow to read various C types (`doubleAt:`, `ucharAt:`, etc.). These are mostly obsoleted by `CObject` which, in newer versions of GNU Smalltalk, supports manually managed heap-backed memory as well as garbage collected ByteArray-backed memory.

Another interesting class is ObjectMemory. This provides a few methods that enable one to tune the virtual machine's usage of memory; many methods that in the past were instance methods of Smalltalk or class methods of Memory are now class methods of ObjectMemory. In addition, and that's what the rest of this section is about, the virtual machines signals events to its dependents exactly through this class.

The events that can be received are

returnFromSnapshot
: This is sent every time an image is restarted, and substitutes the concept of an *init block* that was present in previous versions.

aboutToQuit
: This is sent just before the interpreter is exiting, either because `ObjectMemory quit` was sent or because the specified files were all filed in. Exiting from within this event might cause an infinite loop, so be careful.

aboutToSnapshot
: This is sent just before an image file is created. Exiting from within this event will leave any preexisting image untouched.

finishedSnapshot
: This is sent just after an image file is created. Exiting from within this event will not make the image unusable.

2.9 Memory management in GNU Smalltalk

The GNU Smalltalk virtual machine is equipped with a garbage collector, a facility that reclaims the space occupied by objects that are no longer accessible from the system roots. The collector is composed of several parts, each of which can be invoked by the virtual machine using various tunable strategies, or invoked manually by the programmer.

These parts include a *generation scavenger*, a *mark & sweep* collectory with an incremental sweep phase, and a *compactor*. All these facilities work on different memory spaces and differs from the other in its scope, speed and disadvantages (which are hopefully balanced by the availability of different algorithms). What follows is a description of these algorithms and of the memory spaces they work in.

NewSpace is the memory space where young objects live. It is composed of three subspaces: an object-creation space (*Eden*) and two *SurvivorSpaces*. When an object is first created, it is placed in Eden. When Eden starts to fill up (i.e., when the number of used bytes in Eden exceeds the scavenge threshold), objects that are housed in Eden or in the occupied SurvivorSpace and that are still reachable from the system roots are copied to the unoccupied SurvivorSpace. As an object survives different scavenging passes, it will be

shuffled by the scavenger from the occupied SurvivorSpace to the unoccupied one. When the number of used bytes in SurvivorSpace is high enough that the scavenge pause might be excessively long, the scavenger will move some of the older surviving objects from NewSpace to *OldSpace*. In the garbage collection jargon, we say that such objects are being *tenured* to OldSpace.

This garbage collection algorithm is designed to reclaim short-lived objects, that is those objects that expire while residing in NewSpace, and to decide when enough data is residing in NewSpace that it is useful to move some of it in OldSpace. A *copying* garbage collector is particularly efficient in an object population whose members are more likely to die than survive, because this kind of scavenger spends most of its time copying survivors, who will be few in number in such populations, rather than tracing corpses, who will be many in number. This fact makes copying collection especially well suited to NewSpace, where a percentage of 90% or more objects often fails to survive across a single scavenge.

The particular structure of NewSpace has many advantages. On one hand, having a large Eden and two small SurvivorSpaces has a smaller memory footprint than having two equally big semi-spaces and allocating new objects directly from the occupied one (by default, GNU Smalltalk uses 420=300+60*2 kilobytes of memory, while a simpler configuration would use 720=360*2 kilobytes). On the other hand, it makes tenuring decisions particularly simple: the copying order is such that short-lived objects tend to be copied last, while objects that are being referred from OldSpace tend to be copied first: this is because the tenuring strategy of the scavenger is simply to treat the destination SurvivorSpace as a circular buffer, tenuring objects with a First-In-First-Out policy.

An object might become part of the scavenger root set for several reasons: objects that have been tenured are roots if their data lives in an OldSpace page that has been written to since the last scavenge (more on this later), plus all objects can be roots if they are known to be referenced from C code or from the Smalltalk stacks.

In turn, some of the old objects can be made to live in a special area, called *FixedSpace*. Objects that reside in FixedSpace are special in that their body is guaranteed to remain at a fixed address (in general, GNU Smalltalk only ensures that the header of the object remains at a fixed address in the Object Table). Because the garbage collector can and does move objects, passing objects to foreign code which uses the object's address as a fixed key, or which uses a ByteArray as a buffer, presents difficulties. One can use `CObject` to manipulate C data on the `malloc` heap, which indeed does not move, but this can be tedious and requires the same attentions to avoid memory leaks as coding in C. FixedSpace provides a much more convenient mechanism: once an object is deemed fixed, the object's body will never move through-out its life-time; the space it occupies will however still be returned automatically to the FixedSpace pool when the object is garbage collected. Note that because objects in FixedSpace cannot move, FixedSpace cannot be compacted and can therefore suffer from extensive fragmentation. For this reason, FixedSpace should be used carefully. FixedSpace however is rebuilt (of course) every time an image is brought up, so a kind of compaction of FixedSpace can be achieved by saving a snapshot, quitting, and then restarting the newly saved image.

Memory for OldSpace and FixedSpace is allocated using a variation of the system allocator `malloc`: in fact, GNU Smalltalk uses the same allocator for its own internal needs, for OldSpace and for FixedSpace, but it ensures that a given memory page never hosts objects that reside in separate spaces. New pages are mapped into the address space as

needed and devoted to OldSpace or FixedSpace segments; similarly, when unused they may be subsequently unmapped, or they might be left in place waiting to be reused by `malloc` or by another Smalltalk data space.

Garbage that is created among old objects is taken care of by a mark & sweep collector which, unlike the scavenger which only reclaims objects in NewSpace, can only reclaim objects in OldSpace. Note that as objects are allocated, they will not only use the space that was previously occupied in the Eden by objects that have survived, but they will also reuse the entries in the global Object Table that have been freed by object that the scavenger could reclaim. This quest for free object table entries can be combined with the sweep phase of the OldSpace collector, which can then be done incrementally, limiting the disruptive part of OldSpace garbage collection to the mark phase.

Several runs of the mark & sweep collector can lead to fragmentation (where objects are allocated from several pages, and then become garbage in an order such that a bunch of objects remain in each page and the system is not able to recycle them). For this reason, the system periodically tries to compact OldSpace. It does so simply by looping through every old object and copying it into a new OldSpace. Since the OldSpace allocator does not suffer from fragmentation until objects start to be freed nor after all objects are freed, at the end of the copy all the pages in the fragmented OldSpace will have been returned to the system (some of them might already have been used by the compacted OldSpace), and the new, compacted OldSpace is ready to be used as the system OldSpace. Growing the object heap (which is done when it is found to be quite full even after a mark & sweep collection) automatically triggers a compaction.

You can run the compactor without marking live objects. Since the amount of garbage in OldSpace is usually quite limited, the overhead incurred by copying potentially dead objects is small enough that the compactor still runs considerably faster than a full garbage collection, and can still give the application some breathing room.

Keeping OldSpace and FixedSpace in the same heap would then make compaction of OldSpace (whereby it is rebuilt from time to time in order to limit fragmentation) much less effective. Also, the `malloc` heap is not used for FixedSpace objects because GNU Smalltalk needs to track writes to OldSpace and FixedSpace in order to support efficient scavenging of young objects.

To do so, the grey page table[2] contains one entry for each page in OldSpace or FixedSpace that is thought to contain at least a reference to an object housed in NewSpace. Every page in OldSpace is created as grey, and is considered grey until a scavenging pass finds out that it actually does not contain pointers to NewSpace. Then the page is recolored black[3], and will stay black until it is written to or another object is allocated in it (either a new fixed object, or a young object being tenured). The grey page table is expanded and shrunk as needed by the virtual machine.

---

[2] The denomination *grey* comes from the lexicon of *tri-color marking*, which is an abstraction of every possible garbage collection algorithm: in tri-color marking, grey objects are those that are known to be reachable or that we are not interested in reclaiming, yet have not been scanned to mark the objects that they refer to as reachable.

[3] Black objects are those that are known to be reachable or that we are not interested in reclaiming, and are known to have references only to other black or grey objects (in case you're curious, the tri-color marking algorithm goes on like this: object not yet known to be reachable are white, and when all objects are either black or white, the white ones are garbage).

Drawing an histogram of object sizes shows that there are only a few sources of large objects on average (i.e., objects greater than a page in size), but that enough of these objects are created dynamically that they must be handled specially. Such objects should not be allocated in NewSpace along with ordinary objects, since they would fill up NewSpace prematurely (or might not even fit in it), thus accelerating the scavenging rate, reducing performance and resulting in an increase in tenured garbage. Even though this is not an optimal solution because it effectively tenures these objects at the time they are created, a benefit can be obtained by allocating these objects directly in FixedSpace. The reason why FixedSpace is used is that these objects are big enough that they don't result in fragmentation[4]; and using FixedSpace instead of OldSpace avoids that the compactor copies them because this would not provide any benefit in terms of reduced fragmentation.

Smalltalk activation records are allocated from another special heap, the context pool. This is because it is often the case that they can be deallocated in a Last-In-First-Out (stack) fashion, thereby saving the work needed to allocate entries in the object table for them, and quickly reusing the memory that they use. When the activation record is accessed by Smalltalk, however, the activation record must be turned into a first-class `OOP`[5]. Since even these objects are usually very short-lived, the data is however not copied to the Eden: the eviction of the object bodies from the context pool is delayed to the next scavenging, which will also empty the context pool just like it empties Eden. If few objects are allocated and the context pool turns full before the Eden, a scavenging is also triggered; this is however quite rare.

Optionally, GNU Smalltalk can avoid the overhead of interpretation by executing a given Smalltalk method only after that method has been compiled into the underlying microprocessor's machine code. This machine-code generation is performed automatically, and the resulting machine code is then placed in `malloc`-managed memory. Once executed, a method's machine code is left there for subsequent execution. However, since it would require way too much memory to permanently house the machine-code version of every Smalltalk method, methods might be compiled more than once: when a translation is not used at the time that two garbage collection actions are taken (scavenges and global garbage collections count equally), the incremental sweeper discards it, so that it will be recomputed if and when necessary.

2.10 Security in GNU Smalltalk

2.11 Special kinds of objects

A few methods in Object support the creation of particular objects. This include:

- finalizable objects
- weak and ephemeron objects (i.e. objects whose contents are considered specially, during the heap scanning phase of garbage collection).
- read-only objects (like literals found in methods)

[4] Remember that free pages are shared among the three heaps, that is, OldSpace, FixedSpace and the `malloc` heap. When a large object is freed, the memory that it used can be reused by `malloc` or by OldSpace allocation

[5] This is short for *Ordinary Object Pointer*.

- fixed objects (guaranteed not to move across garbage collections)

They are:

`makeWeak` [Method on `Object`]
: Marks the object so that it is considered weak in subsequent garbage collection passes. The garbage collector will consider dead an object which has references only inside weak objects, and will replace references to such an "almost-dead" object with nils, and then send the `mourn` message to the object.

`makeEphemeron` [Method on `Object`]
: Marks the object so that it is considered specially in subsequent garbage collection passes. Ephemeron objects are sent the message `mourn` when the first instance variable is not referenced or is referenced *only through another instance variable in the ephemeron*.

  Ephemerons provide a very versatile base on which complex interactions with the garbage collector can be programmed (for example, finalization which is described below is implemented with ephemerons).

`addToBeFinalized` [Method on `Object`]
: Marks the object so that, as soon as it becomes unreferenced, its `finalize` method is called. Before `finalize` is called, the VM implicitly removes the objects from the list of finalizable ones. If necessary, the `finalize` method can mark again the object as finalizable, but by default finalization will only occur once.

  Note that a finalizable object is kept in memory even when it has no references, because tricky finalizers might "resuscitate" the object; automatic marking of the object as not to be finalized has the nice side effect that the VM can simply delay the releasing of the memory associated to the object, instead of being forced to waste memory even after finalization happens.

  An object must be explicitly marked as to be finalized *every time the image is loaded*; that is, finalizability is not preserved by an image save. This was done because in most cases finalization is used together with operating system resources that would be stale when the image is loaded again. For `CObject`s, in particular, freeing them would cause a segmentation violation.

`removeToBeFinalized` [Method on `Object`]
: Removes the to-be-finalized mark from the object. As I noted above, the finalize code for the object does not have to do this explicitly.

`finalize` [Method on `Object`]
: This method is called by the VM when there are no more references to the object (or, of course, if it only has references inside weak objects).

`isReadOnly` [Method on `Object`]
: This method answers whether the VM will refuse to make changes to the objects when methods like `become:`, `basicAt:put:`, and possibly `at:put:` too (depending on the implementation of the method). Note that GNU Smalltalk won't try to intercept assignments to fixed instance variables, nor assignments via `instVarAt:put:`. Many objects (Characters, `nil`, `true`, `false`, method literals) are read-only by default.

`makeReadOnly:` *aBoolean* [Method on `Object`]
: Changes the read-only or read-write status of the receiver to that indicated by `aBoolean`.

`basicNewInFixedSpace` [Method on `Object`]
: Same as `#basicNew`, but the object won't move across garbage collections.

`basicNewInFixedSpace:` [Method on `Object`]
: Same as `#basicNew:`, but the object won't move across garbage collections.

`makeFixed` [Method on `Object`]
: Ensure that the receiver won't move across garbage collections. This can be used either if you decide after its creation that an object must be fixed, or if a class does not support using `#new` or `#new:` to create an object

Note that, although particular applications will indeed have a need for fixed, read-only or finalizable objects, the `#makeWeak` primitive is seldom needed and weak objects are normally used only indirectly, through the so called *weak collections*. These are easier to use because they provide additional functionality (for example, `WeakArray` is able to determine whether an item has been garbage collected, and `WeakSet` implements hash table functionality); they are:

- `WeakArray`
- `WeakSet`
- `WeakKeyDictionary`
- `WeakValueLookupTable`
- `WeakIdentitySet`
- `WeakKeyIdentityDictionary`
- `WeakValueIdentityDictionary`

Versions of GNU Smalltalk preceding 2.1 included a `WeakKeyLookupTable` class which has been replaced by `WeakKeyDictionary`; the usage is completely identical, but the implementation was changed to use a more efficient approach based on ephemeron objects.

3 Packages

GNU Smalltalk includes a packaging system which allows one to file in components (often called *goodies* in Smalltalk lore) without caring of whether they need other goodies to be loaded first.

The packaging system is implemented by a Smalltalk class, `PackageLoader`, which looks for information about packages in various places:

- the kernel directory's parent directory; this is where an installed `packages.xml` resides, in a system-wide data directory such as `/usr/local/share/smalltalk`;
- the above directory's `site-packages` subdirectory, for example `/usr/local/share/smalltalk/site-pacl`
- in the file `.st/packages.xml`, hosting per-user packages;
- finally, there can be a `packages.xml` in the same directory as the current image.

Each of this directories can contain package descriptions in an XML file named (guess what) `packages.xml`, as well as standalone packages in files named `*.star` (short for *Smalltalk archive*). Later in this section you will find information about `gst-package`, a program that helps you create `.star` files.

There are two ways to load something using the packaging system. The first way is to use the PackageLoader's `fileInPackage:` and `fileInPackages:` methods. For example:

```
PackageLoader fileInPackages: #('DBD-MySQL' 'DBD-SQLite').
PackageLoader fileInPackage: 'Sockets'.
```

The second way is to use the `gst-load` script which is installed together with the virtual machine. For example, you can do:

```
gst-load DBD-MySQL DBD-SQLite DBI
```

and GNU Smalltalk will automatically file in:

- DBI, loaded first because it is needed by the other two packages
- Sockets and Digest, not specified, but needed by DBD-MySQL
- DBD-MySQL
- DBD-SQLite

Notice how DBI has already been loaded.

Then it will save the Smalltalk image, and finally exit.

`gst-load` supports several options:

`-I`
`--image-file`
: Load the packages inside the given image.

`-i`
`--rebuild-image`
: Build an image from scratch and load the package into it. Useful when the image specified with `-I` does not exist yet.

`-q`
`--quiet`
: Hide the script's output.

`-v`
`--verbose`
: Show which files are loaded, one by one.

`-f`
`--force`
: If a package given on the command-line is already present, reload it. This does not apply to automatically selected prerequisites.

`-t`
`--test`
: Run the package testsuite before installing, and exit with a failure if the tests fail. Currently, the testsuites are placed in the image together with the package, but this may change in future versions.

`-n`

`--dry-run`
: Do not save the image after loading.

`--start[=ARG]`
: Start the services identified by the package. If an argument is given, only one package can be specified on the command-line. If at least one package specifies a startup script, `gst-load` won't exit.

To provide support for this system, you have to give away with your GNU Smalltalk goodies a small file (usually called `package.xml`) which looks like this:

```
<package>
  <name>DBD-SQLite</name>
  <namespace>DBI.SQLite</namespace>

  <!-- The prereq tag identifies packages that
     must be loaded before this one. -->
  <prereq>DBI</prereq>

  <!-- The module tag loads a dynamic shared object
     and calls the gst_initModule function in it. Modules
     can register functions so that Smalltalk code can call them,
     and can interact with or manipulate Smalltalk objects. -->
  <module>dbd-sqlite3</module>

  <!-- A separate subpackage can be defined for testing purposes.
     The SUnit package is implicitly made a prerequisite of the
     testing subpackage, and the default value of namespace
     is the one given for the outer package. -->
  <test>
    <!-- Specifies a testing script that gst-sunit (see Section 3.7 [SUnit], page 41)
       will run in order to test the package. If this is specified outside
       the testing subpackage, the package should list SUnit among
       the prerequisites. -->
    <sunit>DBI.SQLite.SQLiteTestSuite</sunit>
    <filein>SQLiteTests.st</filein>
```

```
    </test>

    <!-- The filein tag identifies files that
       compose this package and that should be loaded in the
       image in this order. -->
    <filein>SQLite.st</filein>
    <filein>Connection.st</filein>
    <filein>ResultSet.st</filein>
    <filein>Statement.st</filein>
    <filein>Row.st</filein>
    <filein>ColumnInfo.st</filein>
    <filein>Table.st</filein>
    <filein>TableColumnInfo.st</filein>

    <!-- The file tag identifies extra files that
       compose this package's distribution. -->
    <file>SQLiteTests.st</file>
    <file>ChangeLog</file>
  </package>
```

Other tags exist:

`url` Specifies a URL at which a repository for the package can be found. The repository, when checked out, should contain a `package.xml` file at its root. The contents of this tag are not used for local packages; they are used when using the `--download` option to `gst-package`.

`library` Loads a dynamic shared object and registers the functions in it so that they can all be called from Smalltalk code. The `GTK` package registers the GTK+ library in this way, so that the bindings can use them.

`callout` Instructs to load the package only if the C function whose name is within the tag is available to be called from Smalltalk code.

`start` Specifies a Smalltalk script that `gst-load` and `gst-remote` will execute in order to start the execution of the service implemented in the package. Before executing the script, `%1` is replaced with either `nil` or a String literal.

`stop` Specifies a Smalltalk script that `gst-remote` will execute in order to shut down the service implemented in the package. Before executing the script, `%1` is replaced with either `nil` or a String literal.

`dir` Should include a `name` attribute. The `file`, `filein` and `built-file` tags that are nested within a `dir` tag are prepended with the directory specified by the attribute.

`test` Specifies a subpackage that is only loaded by `gst-sunit` in order to test the package. The subpackage may include arbitrary tags (including `file`, `filein` and `sunit`) but not `name`.

`provides` In some cases, a single functionality can be provided by multiple modules. For example, GNU Smalltalk includes two browsers but only one should be loaded

at any time. To this end, a dummy package `Browser` is created pointing to the default browser (`VisualGST`), but both browsers use `provides` so that if the old BLOX browser is in the image, loading `Browser` will have no effect.

To install your package, you only have to do

```
gst-package path/to/package.xml
```

`gst-package` is a Smalltalk script which will create a `.star` archive in the current image directory, with the files specified in the `file`, `filein` and `built-file` tags. By default the package is placed in the system-wide package directory; you can use the option `--target-directory` to create the `.star` file elsewhere.

Instead of a local `package.xml` file, you can give:

- a local `.star` file or a `URL` to such a file. The file will be downloaded if necessary, and copied to the target directory;
- a URL to a `package.xml` file. The `url` tag in the file will be used to find a source code repository (`git` or `svn`) or as a redirect to another `package.xml` file.

There is also a short form for specifying `package.xml` file on GNU Smalltalk's web site, so that the following two commands are equivalent:

```
gst-package http://smalltalk.gnu.org/project/Iliad/package.xml
gst-package --download Iliad
```

When downloading remote `package.xml` files, `gst-package` also performs a special check to detect multiple packages in the same repository. If the following conditions are met:

- a package named *`package`* has a prerequisite *`package-subpackage`*;
- there is a toplevel subdirectory *subpackage* in the repository;
- the subdirectory has a `package.xml` file in it

then the *`subpackage`*`/package.xml` will be installed as well. `gst-package` does not check if the file actually defines a package with the correct name, but this may change in future versions.

Alternatively, `gst-package` can be used to create a skeleton GNU style source tree. This includes a `configure.ac` that will find the installation path of GNU Smalltalk, and a `Makefile.am` to support all the standard Makefile targets (including `make install` and `make dist`). To do so, go in the directory that is to become the top of the source tree and type.

```
gst-package --prepare path1/package.xml path2/package.xml
```

In this case the generated configure script and Makefile will use more features of `gst-package`, which are yet to be documented. The GNU Smalltalk makefile similarly uses `gst-package` to install packages and to prepare the distribution tarballs.

The rest of this chapter discusses some of the packages provided with GNU Smalltalk.

3.1 GTK and VisualGST

GNU Smalltalk comes with GTK bindings and with a browser based on it. The system can be started as `gst-browser` and will allow the programmer to view the source code for existing classes, to modify existing classes and methods, to get detailed information about the classes and methods, and to evaluate code within the browser. In addition, simple debugging and

unit testing tools are provided. An Inspector window allows the programmer to graphically inspect and modify the representation of an object and a walkback inspector was designed which will display a backtrace when the program encounters an error. SUnit tests (see Section 3.7 [SUnit], page 41) can be run from the browser in order to easily support test driven development.

The Transcript global object is redirected to print to the transcript window instead of printing to stdout, and the transcript window as well as the workspaces, unlike the console read-eval-print loop, support variables that live across multiple evaluations:

```
a := 2   "Do-it"
a + 2    "Print-it: 4 will be shown"
```

To start the browser you can simply type:

```
gst-browser
```

This will load any requested packages, then, if all goes well, a *launcher* window combining all the basic tools will appear on your display.

3.2 The Smalltalk-in-Smalltalk library

The Smalltalk-in-Smalltalk library is a set of classes for looking at Smalltalk code, constructing models of Smalltalk classes that can later be created for real, analyzing and performing changes to the image, finding smelly code and automatically doing repetitive changes. This package incredibly enhances the reflective capabilities of Smalltalk.

A fundamental part of the system is the recursive-descent parser which creates parse nodes in the form of instances of subclasses of `RBProgramNode`.

The parser's extreme flexibility can be exploited in three ways, all of which are demonstrated by source code available in the distribution:

- First, actions are not hard-coded in the parser itself: the parser creates a parse tree, then hands it to methods in `RBParser` that can be overridden in different `RBParser` subclasses. This is done by the compiler itself, in which a subclass of `RBParser` (class `STFileInParser`) hands the parse trees to the `STCompiler` class.
- Second, an implementation of the "visitor" pattern is provided to help in dealing with parse trees created along the way; this approach is demonstrated by the Smalltalk code pretty-printer in class `RBFormatter`, by the syntax highlighting engine included with the browser, and by the compiler.
- The parser is able to perform complex tree searches and rewrites, through the ParseTreeSearcher and ParseTreeRewriter classes.

In addition, two applications were created on top of this library which are specific to GNU Smalltalk. The first is a compiler for Smalltalk methods written in Smalltalk itself, whose source code provides good insights into the GNU Smalltalk virtual machine.

The second is the automatic documentation extractor. `gst-doc` is able to create documentation even if the library cannot be loaded (for example, if loading it requires a running X server). To do so it uses `STClassLoader` from the `Parser` package to load and interpret Smalltalk source code, creating objects for the classes and methods being read in; then, polymorphism allows one to treat these exactly like usual classes.

3.3 Database connectivity

GNU Smalltalk includes support for connecting to databases. Currently this support is limited to retrieving result sets from SQL selection queries and executing SQL data manipulation queries; in the future however a full object model will be available that hides the usage of SQL.

Classes that are independent of the database management system that is in use reside in package `DBI`, while the drivers proper reside in separate packages which have `DBI` as a prerequisite; currently, drivers are supplied for *MySQL* and *PostgreSQL*, in packages `DBD-MySQL` and `DBD-PostgreSQL` respectively.

Using the library is fairly simple. To execute a query you need to create a connection to the database, create a statement on the connection, and execute your query. For example, let's say I want to connect to the `test` database on the localhost. My user name is `doe` and my password is `mypass`.

```
| connection statement result |

connection := DBI.Connection
    connect: 'dbi:MySQL:dbname=test;hostname=localhost'
    user: 'doe'
    password: 'mypass').
```

You can see that the DBMS-specific classes live in a sub-namespace of `DBI`, while DBMS-independent classes live in `DBI`.

Here is how I execute a query.

```
statement := connection execute: 'insert into aTable (aField) values (123)'.
```

The result that is returned is a `ResultSet`. For write queries the object returns the number of rows affected. For read queries (such as selection queries) the result set supports standard stream protocol (`next`, `atEnd` to read rows off the result stream) and can also supply collection of column information. These are instances of `ColumnInfo`) and describe the type, size, and other characteristics of the returned column.

A common usage of a ResultSet would be:

```
| resultSet values |
[resultSet atEnd] whileFalse: [values add: (resultSet next at: 'columnName') ].
```

3.4 Internationalization and localization support

Different countries and cultures have varying conventions for how to communicate. These conventions range from very simple ones, such as the format for representing dates and times, to very complex ones, such as the language spoken. Provided the programs are written to obey the choice of conventions, they will follow the conventions preferred by the user. GNU Smalltalk provides two packages to ease you in doing so. The `I18N` package covers both *internationalization* and *multilingualization*; the lighter-weight `Iconv` package covers only the latter, as it is a prerequisite for correct internationalization.

Multilingualizing software means programming it to be able to support languages from every part of the world. In particular, it includes understanding multi-byte character sets (such as UTF-8) and Unicode characters whose *code point* (the equivalent of the ASCII

value) is above 127. To this end, GNU Smalltalk provides the `UnicodeString` class that stores its data as 32-bit Unicode values. In addition, `Character` will provide support for all the over one million available code points in Unicode.

Loading the `I18N` package improves this support through the `EncodedStream` class[1], which interprets and transcodes non-ASCII Unicode characters. This support is mostly transparent, because the base classes `Character`, `UnicodeCharacter` and `UnicodeString` are enhanced to use it. Sending `asString` or `printString` to an instance of `Character` and `UnicodeString` will convert Unicode characters so that they are printed correctly in the current locale. For example, '`$<279> printNl`' will print a small Latin letter '`e`' with a dot above, when the `I18N` package is loaded.

Dually, you can convert `String` or `ByteArray` objects to Unicode with a single method call. If the current locale's encoding is UTF-8, '`#[196 151] asUnicodeString`' will return a Unicode string with the same character as above, the small Latin letter '`e`' with a dot above.

The implementation of multilingualization support is not yet complete. For example, methods such as `asLowercase`, `asUppercase`, `isLetter` do not yet recognize Unicode characters.

You need to exercise some care, or your program will be buggy when Unicode characters are used. In particular, Characters must **not** be compared with `==`[2] and should be printed on a Stream with `display:` rather than `nextPut:`.

Also, Characters need to be created with the class method `codePoint:` if you are referring to their Unicode value; `codePoint:` is also the only method to create characters that is accepted by the ANSI Standard for Smalltalk. The method `value:`, instead, should be used if you are referring to a byte in a particular encoding. This subtle difference means that, for example, the last two of the following examples will fail:

```
"Correct.  Use #value: with Strings, #codePoint: with UnicodeString."
String with: (Character value: 65)
String with: (Character value: 128)
UnicodeString with: (Character codePoint: 65)
UnicodeString with: (Character codePoint: 128)

"Correct.  Only works for characters in the 0-127 range, which may
 be considered as defensive programming."
String with: (Character codePoint: 65)

"Dubious, and only works for characters in the 0-127 range.  With
 UnicodeString, probably you always want #codePoint:."
UnicodeString with: (Character value: 65)

"Fails, we try to use a high character in a String"
String with: (Character codePoint: 128)

"Fails, we try to use an encoding in a Unicode string"
```

[1] All the classes mentioned in this section reside in the `I18N` namespace.

[2] Character equality with `=` will be as fast as with `==`.

```
UnicodeString with: (Character value: 128)
```

Internationalizing software, instead, means programming it to be able to adapt to the user's favorite conventions. These conventions can get pretty complex; for example, the user might specify the locale 'espana-castellano' for most purposes, but specify the locale 'usa-english' for currency formatting: this might make sense if the user is a Spanish-speaking American, working in Spanish, but representing monetary amounts in US dollars. You can see that this system is simple but, at the same time, very complete. This manual, however, is not the right place for a thorough discussion of how an user would set up his system for these conventions; for more information, refer to your operating system's manual or to the GNU C library's manual.

GNU Smalltalk inherits from ISO C the concept of a *locale*, that is, a collection of conventions, one convention for each purpose, and maps each of these purposes to a Smalltalk class defined by the `I18N` package, and these classes form a small hierarchy with class `Locale` as its roots:

- `LcNumeric` formats numbers; `LcMonetary` and `LcMonetaryISO` format currency amounts.
- `LcTime` formats dates and times.
- `LcMessages` translates your program's output. Of course, the package can't automatically translate your program's output messages into other languages; the only way you can support output in the user's favorite language is to translate these messages by hand. The package does, though, provide methods to easily handle translations into multiple languages.

Basic usage of the `I18N` package involves a single selector, the question mark (`?`), which is a rarely used yet valid character for a Smalltalk binary message. The meaning of the question mark selector is "How do you say ... under your convention?". You can send `?` to either a specific instance of a subclass of `Locale`, or to the class itself; in this case, rules for the default locale (which is specified via environment variables) apply. You might say, for example, `LcTime ? Date today` or, for example, `germanMonetaryLocale ? account balance`. This syntax can be at first confusing, but turns out to be convenient because of its consistency and overall simplicity.

Here is how `?` works for different classes:

`?` *aString* [Method on `LcTime`]
Format a date, a time or a timestamp (`DateTime` object).

`?` *aString* [Method on `LcNumber`]
Format a number.

`?` *aString* [Method on `LcMonetary`]
Format a monetary value together with its currency symbol.

`?` *aString* [Method on `LcMonetaryISO`]
Format a monetary value together with its ISO currency symbol.

`?` *aString* [Method on `LcMessages`]
Answer an `LcMessagesDomain` that retrieves translations from the specified file.

`?` *aString* [Method on `LcMessagesDomain`]

Retrieve the translation of the given string.[3]

These two packages provides much more functionality, including more advanced formatting options support for Unicode, and conversion to and from several character sets. For more information, refer to Section "Multilingual and international support with Iconv and I18N" in *the* GNU *Smalltalk Library Reference*.

As an aside, the representation of locales that the package uses is exactly the same as the C library, which has many advantages: the burden of mantaining locale data is removed from GNU Smalltalk's mantainers; the need of having two copies of the same data is removed from GNU Smalltalk's users; and finally, uniformity of the conventions assumed by different internationalized programs is guaranteed to the end user.

In addition, the representation of translated strings is the standard MO file format adopted by the GNU `gettext` library.

3.5 The Seaside web framework

Seaside is a framework to build highly interactive web applications quickly, reusably and maintainably. Features of Seaside include callback-based request handling, hierarchical (component-based) page design, and modal session management to easily implement complex workflows.

A simple Seaside component looks like this:

```
Seaside.WAComponent subclass: MyCounter [
    | count |
    MyCounter class >> canBeRoot [ ^true ]

    initialize [
        super initialize.
        count := 0.
    ]
    states [ ^{ self } ]
    renderContentOn: html [
        html heading: count.
        html anchor callback: [ count := count + 1 ]; with: '++'.
        html space.
        html anchor callback: [ count := count - 1 ]; with: '--'.
    ]
]

MyCounter registerAsApplication: 'mycounter'
```

Most of the time, you will run Seaside in a background virtual machine. First of all, you should load the Seaside packages into a new image like this:

```
$ gst-load -iI seaside.im Seaside Seaside-Development Seaside-Examples
```

[3] The `?` method does not apply to the LcMessagesDomain class itself, but only to its instances. This is because LcMessagesDomain is not a subclass of Locale.

Then, you can start Seaside with either of these commands

```
$ gst-load -I seaside.im --start Seaside
$ gst-remote -I seaside.im --daemon --start=Seaside
```

which will start serving pages at `http://localhost:8080/seaside`. The former starts the server in foreground, the latter instead runs a virtual machine that you can control using further invocations of `gst-remote`. For example, you can stop serving Seaside pages, and bring down the server, respectively with these commands:

```
$ gst-remote --stop=Seaside
$ gst-remote --kill
```

3.6 The Swazoo web server

Swazoo (Smalltalk Web Application Zoo) is a free Smalltalk HTTP server supporting both static web serving and a fully-featured web request resolution framework.

The server can be started using

```
$ gst-load --start[=ARG] Swazoo
```

or loaded into a background GNU Smalltalk virtual machine with

```
$ gst-remote --start=Swazoo[:ARG]
```

Usually, the first time you start Swazoo *ARG* is `swazoodemo` (which starts a simple "Hello, World!" servlet) or a path to a configuration file like this one:

```
<Site name: 'hello'; port: 8080>
 <CompositeResource uriPattern: ''/''>
  <HelloWorldResource uriPattern: ''hello.html''>
 </CompositeResource>
</Site>
```

After this initial step, *ARG* can take the following meanings:

- if omitted altogether, all the sites registered on the server are started;
- if a number, all the sites registered on the server on that port are started;
- if a configuration file name, the server configuration is *replaced* with the one loaded from that file;
- if any other string, the site named *ARG* is started.

In addition, a background server can be stopped using

```
$ gst-remote --stop=Swazoo[:ARG]
```

where *ARG* can have the same meanings, except for being a configuration file.

In addition, package `WebServer` implements an older web server engine which is now superseded by Swazoo. It is based on the GPL'ed WikiWorks project. Apart from porting to GNU Smalltalk, a number of changes were made to the code, including refactoring of classes, better aesthetics, authentication support, virtual hosting, and HTTP 1.1 compliance.

3.7 The SUnit testing package

`SUnit` is a framework to write and perform test cases in Smalltalk, originarily written by the father of Extreme Programming[4], Kent Beck. `SUnit` allows one to write the tests and check results in Smalltalk; while this approach has the disadvantage that testers need to be able to write simple Smalltalk programs, the resulting tests are very stable.

What follows is a description of the philosophy of `SUnit` and a description of its usage, excerpted from Kent Beck's paper in which he describes `SUnit`.

3.7.1 Where should you start?

Testing is one of those impossible tasks. You'd like to be absolutely complete, so you can be sure the software will work. On the other hand, the number of possible states of your program is so large that you can't possibly test all combinations.

If you start with a vague idea of what you'll be testing, you'll never get started. Far better to *start with a single configuration whose behavior is predictable.* As you get more experience with your software, you will be able to add to the list of configurations.

Such a configuration is called a *fixture.* Two example fixtures for testing Floats can be `1.0` and `2.0`; two fixtures for testing Arrays can be `#()` and `#(1 2 3)`.

By choosing a fixture you are saying what you will and won't test for. A complete set of tests for a community of objects will have many fixtures, each of which will be tested many ways.

To design a test fixture you have to

- Subclass TestCase
- Add an instance variable for each known object in the fixture
- Override setUp to initialize the variables

3.7.2 How do you represent a single unit of testing?

You can predict the results of sending a message to a fixture. You need to represent such a predictable situation somehow. The simplest way to represent this is interactively. You open an Inspector on your fixture and you start sending it messages. There are two drawbacks to this method. First, you keep sending messages to the same fixture. If a test happens to mess that object up, all subsequent tests will fail, even though the code may be correct.

More importantly, though, you can't easily communicate interactive tests to others. If you give someone else your objects, the only way they have of testing them is to have you come and inspect them.

By representing each predictable situation as an object, each with its own fixture, no two tests will ever interfere. Also, you can easily give tests to others to run. *Represent a predictable reaction of a fixture as a method.* Add a method to TestCase subclass, and stimulate the fixture in the method.

[4] Extreme Programming is a software engineering technique that focuses on team work (to the point that a programmer looks in real-time at what another one is typing), frequent testing of the program, and incremental design.

3.7.3 How do you test for expected results?

If you're testing interactively, you check for expected results directly, by printing and inspecting your objects. Since tests are in their own objects, you need a way to programmatically look for problems. One way to accomplish this is to use the standard error handling mechanism (`#error:`) with testing logic to signal errors:

```
2 + 3 = 5 ifFalse: [self error: 'Wrong answer']
```

When you're testing, you'd like to distinguish between errors you are checking for, like getting six as the sum of two and three, and errors you didn't anticipate, like subscripts being out of bounds or messages not being understood.

There's not a lot you can do about unanticipated errors (if you did something about them, they wouldn't be unanticipated any more, would they?) When a catastrophic error occurs, the framework stops running the test case, records the error, and runs the next test case. Since each test case has its own fixture, the error in the previous case will not affect the next.

The testing framework makes checking for expected values simple by providing a method, `#should:`, that takes a Block as an argument. If the Block evaluates to true, everything is fine. Otherwise, the test case stops running, the failure is recorded, and the next test case runs.

So, you have to *turn checks into a Block evaluating to a Boolean, and send the Block as the parameter to* `#should:`.

In the example, after stimulating the fixture by adding an object to an empty Set, we want to check and make sure it's in there:

```
SetTestCase>>#testAdd
    empty add: 5.
    self should: [empty includes: 5]
```

There is a variant on `TestCase>>#should:`. `TestCase>>#shouldnt:` causes the test case to fail if the Block argument evaluates to true. It is there so you don't have to use `(...) not`.

Once you have a test case this far, you can run it. Create an instance of your TestCase subclass, giving it the selector of the testing method. Send `run` to the resulting object:

```
(SetTestCase selector: #testAdd) run
```

If it runs to completion, the test worked. If you get a walkback, something went wrong.

3.7.4 How do you collect and run many different test cases?

As soon as you have two test cases running, you'll want to run them both one after the other without having to execute two do it's. You could just string together a bunch of expressions to create and run test cases. However, when you then wanted to run "this bunch of cases and that bunch of cases" you'd be stuck.

The testing framework provides an object to represent *a bunch of tests*, `TestSuite`. A `TestSuite` runs a collection of test cases and reports their results all at once. Taking advantage of polymorphism, `TestSuites` can also contain other `TestSuites`, so you can put Joe's tests and Tammy's tests together by creating a higher level suite. *Combine test cases into a test suite.*

```
(TestSuite named: 'Money')
```

```
add: (MoneyTestCase selector: #testAdd);
add: (MoneyTestCase selector: #testSubtract);
run
```

The result of sending `#run` to a `TestSuite` is a `TestResult` object. It records all the test cases that caused failures or errors, and the time at which the suite was run.

All of these objects are suitable for being stored in the image and retrieved. You can easily store a suite, then bring it in and run it, comparing results with previous runs.

3.7.5 Running testsuites from the command line

GNU Smalltalk includes a Smalltalk script to simplify running SUnit test suites. It is called `gst-sunit`. The command-line to `gst-sunit` specifies the packages, files and classes to test:

`-I`
`--image-file`
: Run tests inside the given image.

`-q`
`--quiet`
: Hide the program's output. The results are still communicated with the program's exit code.

`-v`
`--verbose`
: Be more verbose, in particular this will cause `gst-sunit` to write which test is currently being executed.

`-f` *FILE*
`--file=`*FILE*
: Load *FILE* before running the required test cases.

`-p` *PACKAGE*
`--package=`*PACKAGE*
: Load *PACKAGE* and its dependencies, and add *PACKAGE*'s tests to the set of test cases to run.

CLASS
CLASS`*`
: Add *CLASS* to the set of test cases to run. An asterisk after the class name adds all the classes in *CLASS*'s hierarchy. In particular, each selector whose name starts with `test` constitutes a separate test case.

VAR`=`*VALUE*
: Associate variable *VAR* with a value. Variables allow customization of the testing environment. For example, the username with which to access a database can be specified with variables. From within a test, variables are accessible with code like this:

```
TestSuitesScripter variableAt: 'mysqluser' ifAbsent: [ 'root' ]
```

Note that a `#variableAt:` variant does *not* exist, because the testsuite should pick default values in case the variables are not specified by the user.

3.8 Sockets, WebServer, NetClients

GNU Smalltalk includes an almost complete abstraction of the TCP, UDP and IP protocols. Although based on the standard BSD sockets, this library provides facilities such as buffering and preemptive I/O which a C programmer usually has to implement manually.

The distribution includes a few tests (mostly loopback tests that demonstrate both client and server connection), which are class methods in `Socket`. This code should guide you in the process of creating and using both server and client sockets; after creation, sockets behave practically the same as standard Smalltalk streams, so you should not have particular problems. For more information, refer to Section "Network programming with Sockets" in *the GNU Smalltalk Library Reference*. The library is also used by many other packages, including Swazoo and the MySQL driver.

There is also code implementing the most popular Internet protocols: FTP, HTTP, NNTP, SMTP, POP3 and IMAP. These classes, loaded by the `NetClients` package, are derived from multiple public domain and free software packages available for other Smalltalk dialects and ported to GNU Smalltalk. Future version of GNU Smalltalk will include documentation for these as well.

3.9 An XML parser and object model for GNU Smalltalk

The XML parser library for Smalltalk, loaded as package `XML` includes a validating XML parser and Document Object Model. This library is rapidly becoming a standard in the Smalltalk world and a XSLR interpreter based on it is bundled with GNU Smalltalk as well (see packages `XPath` and `XSL`).

Parts of the basic XML package can be loaded independently using packages `XML-DOM`, `XML-SAXParser`, `XML-XMLParser`, `XML-SAXDriver`, `XML-XMLNodeBuilder`.

3.10 Other packages

Various other "minor" packages are provided, typically as examples of writing modules for GNU Smalltalk (see Section 5.1 [Linking your libraries to the virtual machine], page 47). These include:

Complex which adds transparent operations with complex numbers

GDBM which is an interface to the GNU database manager

Digest which provides two easy to use classes to quickly compute cryptographically strong hash values using the MD5 and SHA1 algorithms.

NCurses which provides bindings to *ncurses*

Continuations
which provides more examples and tests for continuations (an advanced feature to support complex control flow).

DebugTools
which provides a way to attach to another Smalltalk process and execute it a bytecode or a method at a time.

4 Smalltalk interface for GNU Emacs

GNU Smalltalk comes with its own Emacs mode for hacking Smalltalk code. It also provides tools for interacting with a running Smalltalk system in an Emacs subwindow.

Emacs will automatically go into Smalltalk mode when you edit a Smalltalk file (one with the extension `.st`).

4.1 Smalltalk editing mode

The GNU Smalltalk editing mode is there to assist you in editing your Smalltalk code. It tries to be smart about indentation and provides a few cooked templates to save you keystrokes.

Since Smalltalk syntax is highly context sensitive, the Smalltalk editing mode will occasionally get confused when you are editing expressions instead of method definitions. In particular, using local variables, thus:

```
| foo |
    foo := 3.
    ^foo squared !
```

will confuse the Smalltalk editing mode, as this might also be a definition the binary operator `|`, with second argument called '`foo`'. If you find yourself confused when editing this type of expression, put a dummy method name before the start of the expression, and take it out when you're done editing, thus:

```
x
| foo |
    foo := 3.
    ^foo squared !
```

4.2 Smalltalk interactor mode

An interesting feature of Emacs Smalltalk is the Smalltalk interactor, which basically allows you run in GNU Emacs with Smalltalk files in one window, and Smalltalk in the other. You can, with a single command, edit and change method definitions in the live Smalltalk system, evaluate expressions, make image snapshots of the system so you can pick up where you left off, file in an entire Smalltalk file, etc. It makes a tremendous difference in the productivity and enjoyment that you'll have when using GNU Smalltalk.

To start up the Smalltalk interactor, you must be running GNU Emacs and in a buffer that's in Smalltalk mode. Then, if you type *`C-c m`*. A second window will appear with GNU Smalltalk running in it.

This window is in most respects like a Shell mode window. You can type Smalltalk expressions to it directly and re-execute previous things in the window by moving the cursor back to the line that contains the expression that you wish to re-execute and typing return.

Notice the status in the mode line (e.g. '`starting-up`', '`idle`', etc). This status will change when you issue various commands from Smalltalk mode.

When you first fire up the Smalltalk interactor, it puts you in the window in which Smalltalk is running. You'll want to switch back to the window with your file in it to explore the rest of the interactor mode, so do it now.

To execute a range of code, mark the region around and type *C-c e*. The expression in the region is sent to Smalltalk and evaluated. The status will change to indicate that the expression is executing. This will work for any region that you create. If the region does not end with an exclamation point (which is syntactically required by Smalltalk), one will be added for you.

There is also a shortcut, *C-c d* (also invokeable as *M-x smalltalk-doit*), which uses a simple heuristic to figure out the start and end of the expression: it searches forward for a line that begins with an exclamation point, and backward for a line that does not begin with space, tab, or the comment character, and sends all the text in between to Smalltalk. If you provide a prefix argument (by typing *C-u C-c d* for instance), it will bypass the heuristic and use the region instead (just like *C-c e* does).

C-c c will compile a method; it uses a similar heuristic to determine the bounds of the method definition. Typically, you'll change a method definition, type *C-c c* and move on to whatever's next. If you want to compile a whole bunch of method definitions, you'll have to mark the entire set of method definitions (from the `methodsFor:` line to the `! !`) as the region and use *C-c e*.

After you've compiled and executed some expressions, you may want to take a snapshot of your work so that you don't have to re-do things next time you fire up Smalltalk. To do this, you use the *C-c s* command, which invokes `ObjectMemory snapshot`. If you invoke this command with a prefix argument, you can specify a different name for the image file, and you can have that image file loaded instead of the default one by using the `-I` flag on the command line when invoking Smalltalk.

You can also evaluate an expression and have the result of the evaluation printed by using the *C-c p* command. Mark the region and use the command.

To file in an entire file (perhaps the one that you currently have in the buffer that you are working on), type *C-c f*. You can type the name of a file to load at the prompt, or just type return and the file associated with the current buffer will be loaded into Smalltalk.

When you're ready to quit using GNU Smalltalk, you can quit cleanly by using the *C-c q* command. If you want to fire up Smalltalk again, or if (heaven forbid) Smalltalk dies on you, you can use the *C-c m* command, and Smalltalk will be reincarnated. Even if it's running, but the Smalltalk window is not visible, *C-c m* will cause it to be displayed right away.

You might notice that as you use this mode, the Smalltalk window will scroll to keep the bottom of the buffer in focus, even when the Smalltalk window is not the current window. This was a design choice that I made to see how it would work. On the whole, I guess I'm pretty happy with it, but I am interested in hearing your opinions on the subject.

5 Interoperability between C and GNU Smalltalk

5.1 Linking your libraries to the virtual machine

A nice thing you can do with GNU Smalltalk is enhancing it with your own goodies. If they're written in Smalltalk only, no problem: getting them to work as packages (see Chapter 3 [Packages], page 31), and to fit in with the GNU Smalltalk packaging system, is likely to be a five-minutes task.

If your goodie is creating a binding to an external C library and you do not need particular glue to link it to Smalltalk (for example, there are no callbacks from C code to Smalltalk code), you can use the `dynamic library linking` system. When using this system, you have to link GNU Smalltalk with the library at run-time using DLD, using either `DLD class>>#addLibrary:` or a `<library>` tag in a `package.xml` file (see Chapter 3 [Packages], page 31). The following line:

```
DLD addLibrary: 'libc'
```

is often used to use the standard C library functions from Smalltalk.

However, if you want to provide a more intimate link between C and Smalltalk, as is the case with for example the GTK bindings, you should use the `dynamic module linking` system. This section explains what to do, taking the Digest library as a guide.

A module is distinguished from a standard shared library because it has a function which Smalltalk calls to initialize the module; the name of this function must be `gst_initModule`. Here is the initialization function used by Digest:

```
void
gst_initModule(proxy)
     VMProxy *proxy;
{
  vmProxy = proxy;
  vmProxy->defineCFunc ("MD5AllocOOP", MD5AllocOOP);
  vmProxy->defineCFunc ("MD5Update", md5_process_bytes);
  vmProxy->defineCFunc ("MD5Final", md5_finish_ctx);

  vmProxy->defineCFunc ("SHA1AllocOOP", SHA1AllocOOP);
  vmProxy->defineCFunc ("SHA1Update", sha1_process_bytes);
  vmProxy->defineCFunc ("SHA1Final", sha1_finish_ctx);
}
```

Note that the `defineCFunc` function is called through a function pointer in `gst_initModule`, and that the value of its parameter is saved in order to use it elsewhere in its code. This is not strictly necessary on many platforms, namely those where the module is effectively *linked with the Smalltalk virtual machine* at run-time; but since some[1] cannot obtain this, for maximum portability you must always call the virtual machine through the proxy and never refer to any symbol which the virtual machine exports. For uniformity, even programs that link with `libgst.a` should not call these

[1] The most notable are AIX and Windows.

functions directly, but through a `VMProxy` exported by `libgst.a` and accessible through the `gst_interpreter_proxy` variable.

Modules are shared libraries; the default directory in which modules are searched for is stored in a `gnu-smalltalk.pc` file that is installed by GNU Smalltalk so that it can be used with `pkg-config`. An Autoconf macro `AM_PATH_GST` is also installed that will put the directory in the `gstmoduledir` Autoconf substitution. When using GNU Automake and Libtool, you can then build modules by including something like this in `Makefile.am`:

```
gstmodule_LTLIBRARIES = libdigest.la
libdigest_la_LDFLAGS = -module -no-undefined ... more flags ...
libdigest_la_SOURCES = ... your source files ...
```

While you can use `DLD class>>#addModule:` to link a module into the virtual machine at run time, usually bindings that require a module are complex enough to be packaged as `.star` files. In this case, you will have to add the name of the module in a package file (see Chapter 3 [Packages], page 31). In this case, the relevant entry in the file will be

```
<package>
  <name>Digest</name>
  <filein>digest.st</filein>
  <filein>md5.st</filein>
  <filein>sha1.st</filein>
  <module>digest</module>

  <test>
   <sunit>MD5Test SHA1Test</sunit>
   <filein>mdtests.st</filein>
  </test>
</package>
```

There is also a third case, in which the bindings are a mixture of code written specially for GNU Smalltalk, and the normal C library. In this case, you can use a combination of dynamic shared libraries and dynamic modules.

To do this, you can specify both `<library>` and `<module>` tags in the `package.xml` file; alternatively, the following functions allow you to call `DLD class>>#addLibrary:` from within a module.

`mst_Boolean dlOpen` (*void *filename, int module*) [Function]
: Open the library pointed to by with *filename* (which need not include an extension), and invoke gst_initModule if it is found in the library. If *module* is false, add the file to the list of libraries that Smalltalk searches for external symbols.

  Return true if the library was found.

`void dlAddSearchDir` (*const char *dir*) [Function]
: Add *dir* at the beginning of the search path of `dlOpen`.

`void dlPushSearchPath` (*void*) [Function]
: Save the current value of the search path for `dlOpen`. This can be used to temporarily add the search path for the libraries added by a module, without affecting subsequent libraries manually opened with the `DLD` class.

`void dlPopSearchPath` (*void*) [Function]
Restore the last saved value of the search path.

5.2 Using the C callout mechanism

To use the C callout mechanism, you first need to inform Smalltalk about the C functions that you wish to call. You currently need to do this in two places: 1) you need to establish the mapping between your C function's address and the name that you wish to refer to it by, and 2) define that function along with how the argument objects should be mapped to C data types to the Smalltalk interpreter. As an example, let us use the pre-defined (to GNU Smalltalk) functions of `system` and `getenv`.

First, the mapping between these functions and string names for the functions needs to be established in your module. If you are writing an external Smalltalk module (which can look at Smalltalk objects and manipulate them), see Section 5.1 [Linking your libraries to the virtual machine], page 47; if you are using function from a dynamically loaded library, see Section 2.6 [Dynamic loading], page 22.

Second, we need to define a method that will invoke these C functions and describe its arguments to the Smalltalk runtime system. Such a method is defined with a primitive-like syntax, similar to the following example (taken from `kernel/CFuncs.st`)

```
system: aString
    <cCall: 'system' returning: #int args: #(#string)>

getenv: aString
    <cCall: 'getenv' returning: #string args: #(#string)>
```

These methods were defined on class `SystemDictionary`, so that we would invoke it thus:

```
Smalltalk system: 'lpr README' !
```

However, there is no special significance to which class receives the method; it could have just as well been Float, but it might look kind of strange to see:

```
1701.0 system: 'mail help-smalltalk@gnu.org' !
```

The various keyword arguments are described below.

`cCall: 'system'`
This says that we are defining the C function `system`. This name must be **exactly** the same as the string passed to `defineCFunc`.

The name of the method does not have to match the name of the C function; we could have just as easily defined the selector to be `'rambo: fooFoo'`; it's just good practice to define the method with a similar name and the argument names to reflect the data types that should be passed.

`returning: #int`
This defines the C data type that will be returned. It is converted to the corresponding Smalltalk data type. The set of valid return types is:

`char`	Single C character value
`string`	A C char *, converted to a Smalltalk string

`stringOut`
: A C char *, converted to a Smalltalk string and then freed.

`symbol`
: A C char *, converted to a Smalltalk symbol

`symbolOut`
: A C char *, converted to a Smalltalk symbol and then freed.

`int`
: A C int value

`uInt`
: A C unsigned int value

`long`
: A C long value

`uLong`
: A C unsigned long value

`double`
: A C double, converted to an instance of FloatD

`longDouble`
: A C long double, converted to an instance of FloatQ

`void`
: No returned value (`self` returned from Smalltalk)

`wchar`
: Single C wide character (`wchar_t`) value

`wstring`
: Wide C string (`wchar_t *`), converted to a UnicodeString

`wstringOut`
: Wide C string (`wchar_t *`), converted to a UnicodeString and then freed

`cObject`
: An anonymous C pointer; useful to pass back to some C function later

`smalltalk`
: An anonymous (to C) Smalltalk object pointer; should have been passed to C at some point in the past or created by the program by calling other public GNU Smalltalk functions (see Section 5.4 [Smalltalk types], page 57).

`ctype`
: You can pass an instance of CType or one of its subclasses (see Section 5.3 [C data types], page 53). In this case the object will be sent `#narrow` before being returned: an example of this feature is given in the experimental Gtk+ bindings.

`args: #(#string)`
: This is an array of symbols that describes the types of the arguments in order. For example, to specify a call to open(2), the arguments might look something like:

```
args: #(#string #int #int)
```

The following argument types are supported; see above for details.

`unknown`
: Smalltalk will make the best conversion that it can guess for this object; see the mapping table below

`boolean`
: passed as `char`, which is promoted to `int`

`char` passed as `char`, which is promoted to `int`

`wchar` passed as `wchar_t`

`string` passed as `char *`

`byteArrayOut`
passed as `char *`. The contents are expected to be overwritten with a new C string, and copied back to the object that was passed on return from the C function

`stringOut`
passed as `char *`, the contents are expected to be overwritten with a new C string, and the object that was passed becomes the new string on return

`wstring` passed as `wchar_t *`

`wstringOut`
passed as `wchar_t *`, the contents are expected to be overwritten with a new C wide string, and the object that was passed becomes the new string on return

`symbol` passed as `char *`

`byteArray`
passed as `char *`, even though may contain NUL's

`int` passed as `int`

`uInt` passed as `unsigned int`

`long` passed as `long`

`uLong` passed as `unsigned long`

`double` passed as `double`

`longDouble`
passed as `long double`

`cObject` C object value passed as `void *`.

Any class with non-pointer indexed instance variables can be passed as a `#cObject`, and GNU Smalltalk will pass the address of the first indexed instance variable. This however should never be done for functions that allocate objects, call back into Smalltalk code or otherwise may cause a garbage collection: after a GC, pointers passed as `#cObject` may be invalidated. In this case, it is safer to pass every object as `#smalltalk`, or to only pass `CObject`s that were returned by a C function previously.

In addition, `#cObject` can be used for function pointers. These are instances of `CCallable` or one of its subclasses. See Section 5.6 [Smalltalk callbacks], page 65, for more information on how to create function pointers for Smalltalk blocks.

`cObjectPtr`
: Pointer to C object value passed as `void **`. The `CObject` is modified on output to reflect the value stored into the passed object.

`smalltalk`
: Pass the object pointer to C. The C routine should treat the value as a pointer to anonymous storage. This pointer can be returned to Smalltalk at some later point in time.

`variadic`
`variadicSmalltalk`
: an Array is expected, each of the elements of the array will be converted like an `unknown` parameter if `variadic` is used, or passed as a raw object pointer for `variadicSmalltalk`.

`self`
`selfSmalltalk`
: Pass the receiver, converting it to C like an `unknown` parameter if `self` is used or passing the raw object pointer for `selfSmalltalk`. Parameters passed this way don't map to the message's arguments, instead they map to the message's receiver.

Table of parameter conversions:

Declared param type	Object type	C parameter type used
boolean	Boolean (True, False)	int
byteArray	ByteArray	char *
cObject	CObject	void *
cObject	ByteArray, etc.	void *
cObjectPtr	CObject	void **
char	Boolean (True, False)	int
char	Character	int (C promotion rule)
char	Integer	int
double	Float	double (C promotion)
longDouble	Float	long double
int	Boolean (True, False)	int
int	Integer	int
uInt	Boolean (True, False)	unsigned int
uInt	Integer	unsigned int
long	Boolean (True, False)	long
long	Integer	long
uLong	Boolean (True, False)	unsigned long
uLong	Integer	unsigned long
smalltalk, selfSmalltalk	anything	OOP
string	String	char *
string	Symbol	char *
stringOut	String	char *
symbol	Symbol	char *
unknown, self	Boolean (True, False)	int

unknown, self	ByteArray	char *
unknown, self	CObject	void *
unknown, self	Character	int
unknown, self	Float	double
unknown, self	Integer	long
unknown, self	String	char *
unknown, self	Symbol	char *
unknown, self	anything else	OOP
variadic	Array	each element is passed according to "unknown"
variadicSmalltalk	Array	each element is passed as an OOP
wchar	Character	wchar_t
wstring	UnicodeString	wchar_t *
wstringOut	UnicodeString	wchar_t *

When your call-out returns `#void`, depending on your application you might consider using *asynchronous call-outs*. These are call-outs that do not suspend the process that initiated them, so the process might be scheduled again, executing the code that follows the call-out, during the execution of the call-out itself. This is particularly handy when writing event loops (the most common place where you call back into Smalltalk) because then *you can handle events that arrive during the handling of an outer event* before the outer event's processing has ended. Depending on your application this might be correct or not, of course. In the future, asynchronous call-outs might be started into a separate thread.

An asynchronous call-out is defined using an alternate primitive-like syntax, `asyncCCall:args:`. Note that the returned value parameter is missing because an asynchronous call-out always returns `nil`.

5.3 The C data type manipulation system

`CType` is a class used to represent C data types themselves (no storage, just the type). There are subclasses called things like `C`*`mumble`*`CType`. The instances can answer their size and alignment. Their `valueType` is the underlying type of data. It's either an integer, which is interpreted by the interpreter as the scalar type, or the underlying element type, which is another `CType` subclass instance.

To make life easier, there are global variables which hold onto instances of `CScalarCType`: they are called `C`*`mumble`*`Type` (like `CIntType`, not like `CIntCType`), and can be used wherever a C datatype is used. If you had an array of strings, the elements would be CStringType's (a specific instance of CScalarCType).

`CObject` is the base class of the instances of C data. It has a subclass called `CScalar`, which has subclasses called `C`*`mumble`*. These subclasses can answer size and alignment information.

Instances of `CObject` can hold a raw C pointer (for example in `malloc`ed heap)), or can delegate their storage to a `ByteArray`. In the latter case, the storage is automatically garbage collected when the `CObject` becomes dead, and the VM checks accesses to make sure they are in bounds. On the other hand, the storage may move, and for this reason

extra care must be put when using this kind of `CObject` with C routines that call back into Smalltalk, or that store the passed pointer somewhere.

Instances of `CObject` can be created in many ways:

- creating an instance with *`class`* `new` initializes the pointer to `NULL`;
- doing *`type`* `new`, where *type* is a `CType` subclass instance, allocates a new instance with `malloc`.
- doing *`type`* `gcNew`, where *type* is a `CType` subclass instance, allocates a new instance backed by garbage-collected storage.

`CStruct` and `CUnion` subclasses are special. First, `new` allocates a new instance with `malloc` instead of initializing the pointer to `NULL`. Second, they support `gcNew` which creates a new instance backed by garbage-collected storage.

`CObject`s created by the C callout mechanism are never backed by garbage-collected storage.

`CObject` and its subclasses represent a pointer to a C object and as such provide the full range of operations supported by C pointers. For example, `+ anInteger` which returns a CObject which is higher in memory by `anInteger` times the size of each item. There is also `-` which acts like `+` if it is given an integer as its parameter. If a CObject is given, it returns the difference between the two pointers. `incr`, `decr`, `incrBy:`, `decrBy:` adjust the string either forward or backward, by either 1 or `n` characters. Only the pointer to the string is changed; the actual characters in the string remain untouched.

CObjects can be divided into two families, scalars and non-scalars, just like C data types. Scalars fetch a Smalltalk object when sent the `value` message, and change their value when sent the `value:` message. Non-scalars do not support these two messages. Non-scalars include instances of `CArray` and subclasses of `CStruct` and `CUnion` (but not `CPtr`).

`CPtr`s and `CArray`s get their underlying element type through a `CType` subclass instance which is associated with the `CArray` or `CPtr` instance.

`CPtr`'s `value` and `value:` method get or change the underlying value that's pointed to. `value` returns another `CObject` corresponding to the pointed value. That's because, for example, a `CPtr` to `long` points to a place in memory where a pointer to long is stored. It is really a `long **` and must be dereferenced twice with `cPtr value value` to get the `long`.

`CString` is a subclass of `CPtr` that answers a Smalltalk `String` when sent `value`, and automatically allocates storage to copy and null-terminate a Smalltalk `String` when sent `value:`. `replaceWith:` replaces the string the instance points to with a new string or `ByteArray`, passed as the argument. Actually, it copies the bytes from the Smalltalk `String` instance aString into the same buffer already pointed to by the `CString`, with a null terminator.

Finally, there are `CStruct` and `CUnion`, which are abstract subclasses of `CObject`[2]. The following will refer to CStruct, but the same considerations apply to CUnion as well, with the only difference that CUnions of course implement the semantics of a C union.

These classes provide direct access to C data structures including

- `long` (unsigned too)
- `short` (unsigned too)

[2] Actually they have a common superclass named `CCompound`.

- `char` (unsigned too) & byte type
- `double`, `long double`, `float`
- `string` (NUL terminated char *, with special accessors)
- arrays of any type
- pointers to any type
- other structs containing any fixed size types

Here is an example struct decl in C:

```
struct audio_prinfo {
    unsigned    channels;
    unsigned    precision;
    unsigned    encoding;
    unsigned    gain;
    unsigned    port;
    unsigned    _xxx[4];
    unsigned    samples;
    unsigned    eof;
    unsigned char       pause;
    unsigned char       error;
    unsigned char       waiting;
    unsigned char       _ccc[3];
    unsigned char       open;
    unsigned char       active;
};

struct audio_info {
    audio_prinfo_t      play;
    audio_prinfo_t      record;
    unsigned    monitor_gain;
    unsigned    _yyy[4];
};
```

And here is a Smalltalk equivalent decision:

```
CStruct subclass: AudioPrinfo [
    <declaration: #( (#sampleRate #uLong)
                     (#channels #uLong)
                     (#precision #uLong)
                     (#encoding #uLong)
                     (#gain #uLong)
                     (#port #uLong)
                     (#xxx (#array #uLong 4))
                     (#samples #uLong)
                     (#eof #uLong)
                     (#pause #uChar)
                     (#error #uChar)
                     (#waiting #uChar)
                     (#ccc (#array #uChar 3))
```

```
                    (#open #uChar)
                    (#active #uChar))>

    <category: 'C interface-Audio'>
]

CStruct subclass: AudioInfo [
    <declaration: #( (#play #{AudioPrinfo} )
                    (#record #{AudioPrinfo} )
                    (#monitorGain #uLong)
                    (#yyy (#array #uLong 4)))>

    <category: 'C interface-Audio'>
]
```

This creates two new subclasses of `CStruct` called `AudioPrinfo` and `AudioInfo`, with the given fields. The syntax is the same as for creating standard subclasses, with the additional metadata `declaration:`. You can make C functions return `CObject`s that are instances of these classes by passing `AudioPrinfo type` as the parameter to the `returning:` keyword.

AudioPrinfo has methods defined on it like:

```
#sampleRate
#channels
#precision
#encoding
```

etc. These access the various data members. The array element accessors (xxx, ccc) just return a pointer to the array itself.

For simple scalar types, just list the type name after the variable. Here's the set of scalars names, as defined in `kernel/CStruct.st`:

```
#long                   CLong
#uLong                  CULong
#ulong                  CULong
#byte                   CByte
#char                   CChar
#uChar                  CUChar
#uchar                  CUChar
#short                  CShort
#uShort                 CUShort
#ushort                 CUShort
#int                    CInt
#uInt                   CUInt
#uint                   CUInt
#float                  CFloat
#double                 CDouble
#longDouble             CLongDouble
#string                 CString
#smalltalk              CSmalltalk
```

`#{...}`	A given subclass of `CObject`

The `#{...}` syntax is not in the Blue Book, but it is present in GNU Smalltalk and other Smalltalks; it returns an Association object corresponding to a global variable.

To have a pointer to a type, use something like:

```
(#example (#ptr #long))
```

To have an array pointer of size *size*, use:

```
(#example (#array #string size))
```

Note that this maps to `char *example[size]` in C.

The objects returned by using the fields are CObjects; there is no implicit value fetching currently. For example, suppose you somehow got ahold of an instance of class AudioPrinfo as described above (the instance is a CObject subclass and points to a real C structure somewhere). Let's say you stored this object in variable `audioInfo`. To get the current gain value, do

```
audioInfo gain value
```

to change the gain value in the structure, do

```
audioInfo gain value: 255
```

The structure member message just answers a `CObject` instance, so you can hang onto it to directly refer to that structure member, or you can use the `value` or `value:` methods to access or change the value of the member.

Note that this is the same kind of access you get if you use the `addressAt:` method on CStrings or CArrays or CPtrs: they return a CObject which points to a C object of the right type and you need to use `value` and `value:` to access and modify the actual C variable.

5.4 Manipulating Smalltalk data from C

GNU Smalltalk internally maps every object except Integers to a data structure named an *OOP* (which is short for *Ordinary Object Pointer*). An OOP is a pointer to an internal data structure; this data structure basically adds a level of indirection in the representation of objects, since it contains

- a pointer to the actual object data
- a bunch of flags, most of which interest the garbage collection process

This additional level of indirection makes garbage collection very efficient, since the collector is free to move an object in memory without updating every reference to that object in the heap, thereby keeping the heap fully compact and allowing very fast allocation of new objects. However, it makes C code that wants to deal with objects even more messy than it would be without; if you want some examples, look at the hairy code in GNU Smalltalk that deals with processes.

To shield you as much as possible from the complications of doing object-oriented programming in a non-object-oriented environment like C, GNU Smalltalk provides friendly functions to map between common Smalltalk objects and C types. This way you can simply declare OOP variables and then use these functions to treat their contents like C data.

These functions are passed to a module via the `VMProxy` struct, a pointer to which is passed to the module, as shown in Section 5.1 [Linking your libraries to the virtual machine],

page 47. They can be divided in two groups, those that map *from Smalltalk objects to C data types* and those that map *from C data types to Smalltalk objects*.

Here are those in the former group (Smalltalk to C); you can see that they all begin with `OOPTo`:

`long OOPToInt (`*OOP*`)` [Function]
This function assumes that the passed OOP is an Integer and returns the C `signed long` for that integer.

`long OOPToId (`*OOP*`)` [Function]
This function returns an unique identifier for the given OOP, valid until the OOP is garbage-collected.

`double OOPToFloat (`*OOP*`)` [Function]
This function assumes that the passed OOP is an Integer or Float and returns the C `double` for that object.

`long double OOPToLongDouble (`*OOP*`)` [Function]
This function assumes that the passed OOP is an Integer or Float and returns the C `long double` for that object.

`int OOPToBool (`*OOP*`)` [Function]
This function returns a C integer which is true (i.e. `!= 0`) if the given OOP is the `true` object, false (i.e. `== 0`) otherwise.

`char OOPToChar (`*OOP*`)` [Function]
This function assumes that the passed OOP is a Character and returns the C `char` for that integer.

`wchar_t OOPToWChar (`*OOP*`)` [Function]
This function assumes that the passed OOP is a Character or UnicodeCharacter and returns the C `wchar_t` for that integer.

`char *OOPToString (`*OOP*`)` [Function]
This function assumes that the passed OOP is a String or ByteArray and returns a C null-terminated `char *` with the same contents. It is the caller's responsibility to free the pointer and to handle possible '`NUL`' characters inside the Smalltalk object.

`wchar_t *OOPToWString (`*OOP*`)` [Function]
This function assumes that the passed OOP is a UnicodeString and returns a C null-terminated `wchar_t *` with the same contents. It is the caller's responsibility to free the pointer and to handle possible '`NUL`' characters inside the Smalltalk object.

`char *OOPToByteArray (`*OOP*`)` [Function]
This function assumes that the passed OOP is a String or ByteArray and returns a C `char *` with the same contents, without null-terminating it. It is the caller's responsibility to free the pointer.

`PTR OOPToCObject` (*OOP*) [Function]
: This functions assumes that the passed OOP is a kind of CObject and returns a C `PTR` to the C data pointed to by the object. The caller should not free the pointer, nor assume anything about its size and contents, unless it **exactly** knows what it's doing. A `PTR` is a `void *` if supported, or otherwise a `char *`.

`long OOPToC` (*OOP*) [Function]
: This functions assumes that the passed OOP is a String, a ByteArray, a CObject, or a built-in object (`nil`, `true`, `false`, character, integer). If the OOP is `nil`, it answers 0; else the mapping for each object is exactly the same as for the above functions. Note that, even though the function is declared as returning a `long`, you might need to cast it to either a `char *` or `PTR`.

While special care is needed to use the functions above (you will probably want to know at least the type of the Smalltalk object you're converting), the functions below, which convert C data to Smalltalk objects, are easier to use and also put objects in the incubator so that they are not swept by a garbage collection (see Section 5.10 [Incubator], page 74). These functions all *end* with `ToOOP`, except `cObjectToTypedOOP`:

`OOP intToOOP` (*long*) [Function]
: This object returns a Smalltalk `Integer` which contains the same value as the passed C `long`.

`OOP uintToOOP` (*unsigned long*) [Function]
: This object returns a Smalltalk `Integer` which contains the same value as the passed C `unsigned long`.

`OOP idToOOP` (*OOP*) [Function]
: This function returns an OOP from a unique identifier returned by `OOPToId`. The OOP will be the same that was passed to `OOPToId` only if the original OOP has not been garbage-collected since the call to `OOPToId`.

`OOP floatToOOP` (*double*) [Function]
: This object returns a Smalltalk `FloatD` which contains the same value as the passed `double`. Unlike Integers, FloatDs have exactly the same precision as C doubles.

`OOP longDoubleToOOP` (*long double*) [Function]
: This object returns a Smalltalk `FloatQ` which contains the same value as the passed `long double`. Unlike Integers, FloatQs have exactly the same precision as C long doubles.

`OOP boolToOOP` (*int*) [Function]
: This object returns a Smalltalk `Boolean` which contains the same boolean value as the passed C `int`. That is, the returned OOP is the sole instance of either `False` or `True`, depending on where the parameter is zero or not.

`OOP charToOOP` (*char*) [Function]
: This object returns a Smalltalk `Character` which represents the same char as the passed C `char`.

`OOP charToOOP` (*wchar_t*) [Function]
This object returns a Smalltalk `Character` or `UnicodeCharacter` which represents the same char as the passed C `wchar_t`.

`OOP classNameToOOP` (*char **) [Function]
This method returns the Smalltalk class (i.e. an instance of a subclass of Class) whose name is the given parameter. Namespaces are supported; the parameter must give the complete path to the class starting from the `Smalltalk` dictionary. `NULL` is returned if the class is not found.

This method is slow; you can safely cache its result.

`OOP stringToOOP` (*char **) [Function]
This method returns a String which maps to the given null-terminated C string, or the builtin object `nil` if the parameter points to address 0 (zero).

`OOP wstringToOOP` (*wchar_t **) [Function]
This method returns a UnicodeString which maps to the given null-terminated C wide string, or the builtin object `nil` if the parameter points to address 0 (zero).

`OOP byteArrayToOOP` (*char *, int*) [Function]
This method returns a ByteArray which maps to the bytes that the first parameters points to; the second parameter gives the size of the ByteArray. The builtin object `nil` is returned if the first parameter points to address 0 (zero).

`OOP symbolToOOP` (*char **) [Function]
This method returns a String which maps to the given null-terminated C string, or the builtin object `nil` if the parameter points to address 0 (zero).

`OOP cObjectToOOP` (*PTR*) [Function]
This method returns a CObject which maps to the given C pointer, or the builtin object `nil` if the parameter points to address 0 (zero). The returned value has no precise CType assigned. To assign one, use `cObjectToTypedOOP`.

`OOP cObjectToTypedOOP` (*PTR, OOP*) [Function]
This method returns a CObject which maps to the given C pointer, or the builtin object `nil` if the parameter points to address 0 (zero). The returned value has the second parameter as its type; to get possible types you can use `typeNameToOOP`.

`OOP typeNameToOOP` (*char **) [Function]
All this method actually does is evaluating its parameter as Smalltalk code; so you can, for example, use it in any of these ways:

```
cIntType = typeNameToOOP("CIntType");
myOwnCStructType = typeNameToOOP("MyOwnCStruct type");
```

This method is primarily used by `msgSendf` (see Section 5.5 [Smalltalk callin], page 61), but it can be useful if you use lower level call-in methods. This method is slow too; you can safely cache its result.

As said above, the C to Smalltalk layer automatically puts the objects it creates in the incubator which prevents objects from being collected as garbage. A plugin, however, has limited control on the incubator, and the incubator itself is not at all useful when objects should be kept registered for a relatively long time, and whose lives in the registry typically overlap.

To avoid garbage collection of such object, you can use these functions, which access a separate registry:

`OOP registerOOP` (*OOP*) [Function]
Puts the given OOP in the registry. If you register an object multiple times, you will need to unregister it the same number of times. You may want to register objects returned by Smalltalk call-ins.

`void unregisterOOP` (*OOP*) [Function]
Removes an occurrence of the given OOP from the registry.

`void registerOOPArray` (*OOP **, *OOP ***) [Function]
Tells the garbage collector that an array of objects must be made part of the root set. The two parameters point indirectly to the base and the top of the array; that is, they are pointers to variables holding the base and the top of the array: having indirect pointers allows you to dynamically change the size of the array and even to relocate it in memory without having to unregister and re-register it every time you modify it. If you register an array multiple times, you will need to unregister it the same number of times.

`void unregisterOOPArray` (*OOP ***) [Function]
Removes the array with the given base from the registry.

5.5 Calls from C to Smalltalk

GNU Smalltalk provides seven different function calls that allow you to call Smalltalk methods in a different execution context than the current one. The priority in which the method will execute will be the same as the one of Smalltalk process which is currently active.

Four of these functions are more low level and are more suited when the Smalltalk program itself gave a receiver, a selector and maybe some parameters; the others, instead, are more versatile. One of them (`msgSendf`) automatically handles most conversions between C data types and Smalltalk objects, while the others takes care of compiling full snippets of Smalltalk code.

All these functions handle properly the case of specifying, say, 5 arguments for a 3-argument selector—see the description of the single functions for more information).

In all cases except `msgSendf`, passing NULL as the selector will expect the receiver to be a block and evaluate it.

`OOP msgSend` (*OOP receiver, OOP selector, . . .*) [Function]
This function sends the given selector (should be a Symbol, otherwise `nilOOP` is returned) to the given receiver. The message arguments should also be OOPs (otherwise, an access violation exception is pretty likely) and are passed in a NULL-terminated list after the selector. The value returned from the method is passed back

as an OOP to the C program as the result of `msgSend`, or `nilOOP` if the number of arguments is wrong. Example (same as `1 + 2`):

```
OOP shouldBeThreeOOP = vmProxy->msgSend(
    intToOOP(1),
    symbolToOOP("+"),
    intToOOP(2),
    NULL);
```

`OOP strMsgSend` (*OOP receiver, char *selector, ...*) [Function]

This function is the same as above, but the selector is passed as a C string and is automatically converted to a Smalltalk symbol.

Theoretically, this function is a bit slower than `msgSend` if your program has some way to cache the selector and avoiding a call to `symbolToOOP` on every call-in. However, this is not so apparent in "real" code because the time spent in the Smalltalk interpreter will usually be much higher than the time spent converting the selector to a Symbol object. Example:

```
OOP shouldBeThreeOOP = vmProxy->strMsgSend(
    intToOOP(1),
    "+",
    intToOOP(2),
    NULL);
```

`OOP vmsgSend` (*OOP receiver, OOP selector, OOP *args*) [Function]

This function is the same as msgSend, but accepts a pointer to the NULL-terminated list of arguments, instead of being a variable-arguments functions. Example:

```
OOP arguments[2], shouldBeThreeOOP;
arguments[0] = intToOOP(2);
arguments[1] = NULL;
/* ... some more code here ... */

shouldBeThreeOOP = vmProxy->vmsgSend(
    intToOOP(1),
    symbolToOOP("+"),
    arguments);
```

`OOP nvmsgSend` (*OOP receiver, OOP selector, OOP *args, int nargs*) [Function]

This function is the same as msgSend, but accepts an additional parameter containing the number of arguments to be passed to the Smalltalk method, instead of relying on the NULL-termination of args. Example:

```
OOP argument, shouldBeThreeOOP;
argument = intToOOP(2);
/* ... some more code here ... */

shouldBeThreeOOP = vmProxy->nvmsgSend(
    intToOOP(1),
    symbolToOOP("+"),
    &argument,
```

```
                1);
```

`OOP performWith` (*OOP*, *OOP*, *OOP*) [Function]
: Shortcut function to invoke a one-argument selector. The first parameter is the receiver, the second is the selector, the third is the sole argument.

`OOP invokeHook` (*int*) [Function]
: Calls into Smalltalk to process a `ObjectMemory` hook given by the parameter. In practice, `changed:` is sent to `ObjectMemory` with a symbol derived from the parameter. The parameter can be one of:

  - `GST_BEFORE_EVAL`
  - `GST_AFTER_EVAL`
  - `GST_ABOUT_TO_QUIT`
  - `GST_RETURN_FROM_SNAPSHOT`
  - `GST_ABOUT_TO_SNAPSHOT`
  - `GST_FINISHED_SNAPSHOT`

  All cases where the last three should be used should be covered in GNU Smalltalk's source code. The first three, however, can actually be useful in user code.

The two functions that directly accept Smalltalk code are named `evalCode` and `evalExpr`, and they're basically the same. They both accept a single parameter, a pointer to the code to be submitted to the parser. The main difference is that `evalCode` discards the result, while `evalExpr` returns it to the caller as an OOP.

`msgSendf`, instead, has a radically different syntax. Let's first look at some examples.

```
/* 1 + 2 */
int shouldBeThree;
vmProxy->msgSendf(&shouldBeThree, "%i %i + %i", 1, 2)

/* aCollection includes: 'abc' */
OOP aCollection;
int aBoolean;
vmProxy->msgSendf(&aBoolean, "%b %o includes: %s", aCollection, "abc")

/* 'This is a test' printNl -- in two different ways */
vmProxy->msgSendf(NULL, "%v %s printNl", "This is a test");
vmProxy->msgSendf(NULL, "%s %s printNl", "This is a test");

/* 'This is a test', ' ok?' */
char *str;
vmProxy->msgSendf(&str, "%s %s , %s", "This is a test", " ok?");
```

As you can see, the parameters to msgSendf are, in order:

- A pointer to the variable which will contain the record. If this pointer is `NULL`, it is discarded.

- A description of the method's interface in this format (the object types, after percent signs, will be explained later in this section)

```
%result_type %receiver_type selector %param1_type %param2_type
```

- A C variable or Smalltalk object (depending on the type specifier) for the receiver
- If needed, the C variables and/or Smalltalk object (depending on the type specifiers) for the arguments.

Note that the receiver and parameters are NOT registered in the object registry (see Section 5.4 [Smalltalk types], page 57). *receiver_type* and *paramX_type* can be any of these characters, with these meanings:

Specifier	C data type	equivalent Smalltalk class
i	long	Integer (see intToOOP)
f	double	Float (see floatToOOP)
F	long double	Float (see longDoubleToOOP)
b	int	True or False (see boolToOOP)
B	OOP	BlockClosure
c	char	Character (see charToOOP)
C	PTR	CObject (see cObjToOOP)
s	char *	String (see stringToOOP)
S	char *	Symbol (see symbolToOOP)
o	OOP	any
t	char *, PTR	CObject (see below)
T	OOP, PTR	CObject (see below)
w	wchar_t	Character (see wcharToOOP)
W	wchar_t *	UnicodeString (see wstringToOOP)

'`%t`' and '`%T`' are particular in the sense that you need to pass *two* additional arguments to `msgSendf`, not one. The first will be a description of the type of the CObject to be created, the second instead will be the CObject's address. If you specify '`%t`', the first of the two arguments will be converted to a Smalltalk `CType` via `typeNameToOOP` (see Section 5.4 [Smalltalk types], page 57); instead, if you specify '`%T`', you will have to directly pass an OOP for the new CObject's type.

For '`%B`' you should not pass a selector, and the block will be evaluated.

The type specifiers you can pass for *result_type* are a bit different:

Specifier	Result if nil	C data type	expected result
i	0L	long	nil or an Integer
f	0.0	double	nil or a Float
F	0.0	long double	nil or a Float
b	0	int	nil or a Boolean
c	'\0'	char	nil or a Character
C	NULL	PTR	nil or a CObject
s	NULL	char *	nil, a String, or a Symbol
?	0	char *, PTR	See oopToC
o	nilOOP	OOP	any (result is not converted)
w	'\0'	wchar_t	nil or a Character
W	NULL	wchar_t *	nil or a UnicodeString

```
v                       /           any (result is discarded)
```

Note that, if resultPtr is `NULL`, the *result_type* is always treated as '`%v`'. If an error occurs, the value in the 'result if nil' column is returned.

5.6 Smalltalk blocks as C function pointers

The Smalltalk callin mechanism can be used effectively to construct bindings to C libraries that require callbacks into Smalltalk. However, it is a "static" mechanism, as the callback functions passed to the libraries have to be written in C and their type signatures are fixed.

If the signatures of the callbacks are not known in advance, and the only way to define callbacks is via C function pointers (as opposed to reflective mechanisms such as the ones in GTK+), then the `VMProxy` functions for Smalltalk callin are not enough.

GNU Smalltalk provides a more dynamic way to convert Smalltalk blocks into C function pointers through the `CCallbackDescriptor` class. This class has a constructor method that is similar to the `cCall:` annotation used for callouts. The method is called `for:returning:withArgs:` and its parameters are:

- a block, whose number of arguments is variable
- a symbol representing the return type
- an array representing the type of the arguments.

The array passed as the third parameter represents values that are passed *from C to Smalltalk* and, as such, should be filled with the same rules that are used by the *return type* of a C callout. In particular, if the C callback accepts an `int *` it is possible (and indeed useful) to specify the type of the argument as `#{CInt}`, so that the block will receive a `CInt` object.

Here is an example of creating a callback which is passed to `glutReshapeFunc`[3]. The desired signature in C is `void (*) (int, int)`.

```
| glut |
...
glut glutReshapeFunc: (CCallbackDescriptor
    for: [ :x :y | self reshape: x@y ]
    returning: #void
    withArgs: #(#int #int))
```

It is important to note that this kind of callback does not survive across an image load (this restriction may be lifted in a future version). When the image is loaded, it has to be reset by sending it the `link` message before it is passed to any C function. Sending the `link` message to an already valid callback is harmless and cheap.

5.7 Other functions available to modules

In addition to the functions described so far, the `VMProxy` that is available to modules contains entry-points for many functions that aid in developing GNU Smalltalk extensions in C. This node documents these functions and the macros that are defined by `libgst/gstpub.h`.

[3] The GLUT bindings use a different scheme for setting up callbacks.

`void asyncCall` (*void (*) (OOP), OOP*) [Function]
This functions accepts a function pointer and an OOP (or `NULL`, but not an arbitrary pointer) and sets up the interpreter to call the function as soon as the next message send is executed.

Caution: This and the next two are the only functions in the `intepreterProxy` that are thread-safe.

`void asyncSignal` (*OOP*) [Function]
This functions accepts an OOP for a `Semaphore` object and signals that object so that one of the processes waiting on that semaphore is waken up. Since a Smalltalk call-in is not an atomic operation, the correct way to signal a semaphore is not to send the `signal` method to the object but, rather, to use:

```
asyncSignal(semaphoreOOP)
```

The signal request will be processed as soon as the next message send is executed.

`void asyncSignalAndUnregister` (*OOP*) [Function]
This functions accepts an OOP for a `Semaphore` object and signals that object so that one of the processes waiting on that semaphore is waken up; the signal request will be processed as soon as the next message send is executed. The object is then removed from the registry.

`void wakeUp` (*void*) [Function]
When no Smalltalk process is running, GNU Smalltalk tries to limit CPU usage by pausing until it gets a signal from the OS. `wakeUp` is an alternative way to wake up the main Smalltalk loop. This should rarely be necessary, since the above functions already call it automatically.

`void syncSignal` (*OOP, mst_Boolean*) [Function]
This functions accepts an OOP for a `Semaphore` object and signals that object so that one of the processes waiting on that semaphore is waken up. If the semaphore has no process waiting in the queue and the second argument is true, an excess signal is added to the semaphore. Since a Smalltalk call-in is not an atomic operation, the correct way to signal a semaphore is not to send the `signal` or `notify` methods to the object but, rather, to use:

```
syncSignal(semaphoreOOP, true)
```

The `sync` in the name of this function distinguishes it from `asyncSignal`, in that it can only be called from a procedure already scheduled with `asyncCall`. It cannot be called from a call-in, or from other threads than the interpreter thread.

`void syncWait` (*OOP*) [Function]
This function is present for backwards-compatibility only and should not be used.

`void showBacktrace` (*FILE **) [Function]
This functions show a backtrace on the given file.

`OOP objectAlloc` (*OOP, int*) [Function]
The `objectAlloc` function allocates an OOP for a newly created instance of the class whose OOP is passed as the first parameter; if that parameter is not a class the results

are undefined (for now, read as "the program will most likely core dump", but that could change in a future version).

The second parameter is used only if the class is an indexable one, otherwise it is discarded: it contains the number of indexed instance variables in the object that is going to be created. Simple uses of `objectAlloc` include:

```
OOP myClassOOP;
OOP myNewObject;
myNewObjectData obj;
...
myNewObject = objectAlloc(myClassOOP, 0);
obj = (myNewObjectData) OOP_TO_OBJ (myNewObject);
obj->arguments = objectAlloc(classNameToOOP("Array"), 10);
...
```

`size_t OOPSize` (*OOP*) [Function]
Return the number of indexed instance variables in the given object.

`OOP OOPAt` (*OOP, size_t*) [Function]
Return an indexed instance variable of the given object. The index is in the second parameter and is zero-based. The function aborts if the index is out of range.

`OOP OOPAtPut` (*OOP, size_t, OOP*) [Function]
Put the object given as the third parameter into an indexed instance variable of the object given as the first parameter. The index in the second parameter and is zero-based. The function aborts if the index is out of range.

The function returns the old value of the indexed instance variable.

`enum gst_indexed_kind OOPIndexedKind` (*OOP*) [Function]
Return the kind of indexed instance variables that the given object has.

`void * OOPIndexedBase` (*OOP*) [Function]
Return a pointer to the first indexed instance variable of the given object. The program should first retrieve the kind of data using OOPIndexedKind.

`OOP getObjectClass` (*OOP*) [Function]
Return the class of the Smalltalk object passed as a parameter.

`OOP getSuperclass` (*OOP*) [Function]
Return the superclass of the class given by the Smalltalk object, that is passed as a parameter.

`mst_Boolean classIsKindOf` (*OOP, OOP*) [Function]
Return true if the class given as the first parameter, is the same or a superclass of the class given as the second parameter.

`mst_Boolean objectIsKindOf` (*OOP, OOP*) [Function]
Return true if the object given as the first parameter is an instance of the class given as the second parameter, or of any of its subclasses.

`mst_Boolean classImplementsSelector` (*OOP, OOP*) [Function]
Return true if the class given as the first parameter implements or overrides the method whose selector is given as the second parameter.

`mst_Boolean classCanUnderstand` (*OOP, OOP*) [Function]
Return true if instances of the class given as the first parameter respond to the message whose selector is given as the second parameter.

`mst_Boolean respondsTo` (*OOP, OOP*) [Function]
Return true if the object given as the first parameter responds to the message whose selector is given as the second parameter.

Finally, several slots of the interpreter proxy provide access to the system objects and to the most important classes. These are:

- `nilOOP`, `trueOOP`, `falseOOP`, `processorOOP`
- `objectClass`, `arrayClass`, `stringClass`, `characterClass`, `smallIntegerClass`, `floatDClass`, `floatEClass`, `byteArrayClass`, `objectMemoryClass`, `classClass`, `behaviorClass`, `blockClosureClass`, `contextPartClass`, `blockContextClass`, `methodContextClass`, `compiledMethodClass`, `compiledBlockClass`, `fileDescriptorClass`, `fileStreamClass`, `processClass`, `semaphoreClass`, `cObjectClass`

More may be added in the future

The macros are[4]:

`gst_object` *OOP_TO_OBJ* (*OOP*) [Macro]
Dereference a pointer to an OOP into a pointer to the actual object data (see Section 5.8 [Object representation], page 69). The result of `OOP_TO_OBJ` is not valid anymore if a garbage-collection happens; for this reason, you should assume that a pointer to object data is not valid after doing a call-in, calling `objectAlloc`, and caling any of the "C to Smalltalk" functions (see Section 5.4 [Smalltalk types], page 57).

`OOP` *OOP_CLASS* (*OOP*) [Macro]
Return the OOP for the class of the given object. For example, `OOP_CLASS(proxy->stringToOOP("Wonderful GNU Smalltalk"))` is the `String` class, as returned by `classNameToOOP("String")`.

`mst_Boolean` *IS_INT* (*OOP*) [Macro]
Return a Boolean indicating whether or not the OOP is an Integer object; the value of SmallInteger objects is encoded directly in the OOP, not separately in a `gst_object` structure. It is not safe to use `OOP_TO_OBJ` and `OOP_CLASS` if `isInt` returns false.

`mst_Boolean` *IS_OOP* (*OOP*) [Macro]
Return a Boolean indicating whether or not the OOP is a 'real' object (and not a SmallInteger). It is safe to use `OOP_TO_OBJ` and `OOP_CLASS` only if `IS_OOP` returns true.

[4] IS_NIL and IS_CLASS have been removed because they are problematic in shared libraries (modules), where they caused undefined symbols to be present in the shared library. These are now private to `libgst.a`. You should use the `nilOOP` field of the interpreter proxy, or `getObjectClass`.

`mst_Boolean` *ARRAY_OOP_AT* (*gst_object, int*) [Macro]
Access the character given in the second parameter of the given Array object. Note that this is necessary because of the way `gst_object` is defined, which prevents `indexedOOP` from working.

`mst_Boolean` *STRING_OOP_AT* (*gst_object, int*) [Macro]
Access the character given in the second parameter of the given String or ByteArray object. Note that this is necessary because of the way `gst_object` is defined, which prevents `indexedByte` from working.

`mst_Boolean` *INDEXED_WORD* (*some-object-type, int*) [Macro]
Access the given indexed instance variable in a `variableWordSubclass`. The first parameter must be a structure declared as described in Section 5.8 [Object representation], page 69).

`mst_Boolean` *INDEXED_BYTE* (*some-object-type, int*) [Macro]
Access the given indexed instance variable in a `variableByteSubclass`. The first parameter must be a structure declared as described in Section 5.8 [Object representation], page 69).

`mst_Boolean` *INDEXED_OOP* (*some-object-type, int*) [Macro]
Access the given indexed instance variable in a `variableSubclass`. The first parameter must be a structure declared as described in Section 5.8 [Object representation], page 69).

5.8 Manipulating instances of your own Smalltalk classes from C

Although GNU Smalltalk's library exposes functions to deal with instances of the most common base class, it's likely that, sooner or later, you'll want your C code to directly deal with instances of classes defined by your program. There are three steps in doing so:

- Defining the Smalltalk class
- Defining a C `struct` that maps the representation of the class
- Actually using the C struct

In this chapter you will be taken through these steps considering the hypotetical task of defining a Smalltalk interface to an SQL server.

The first part is also the simplest, since defining the Smalltalk class can be done in a single way which is also easy and very practical; just evaluate the standard Smalltalk code that does that:

```
Object subclass: SQLAction [
     | database request |
     <category: 'SQL-C interface'>
]

SQLAction subclass: SQLRequest [
     | returnedRows |
     <category: 'SQL-C interface'>
```

```
]
```

To define the C `struct` for a class derived from Object, GNU Smalltalk's `gstpub.h` include file defines an `OBJ_HEADER` macro which defines the fields that constitute the header of every object. Defining a `struct` for SQLAction results then in the following code:

```
struct st_SQLAction {
    OBJ_HEADER;
    OOP database;
    OOP request;
}
```

The representation of SQLRequest in memory is this:

```
.------------------------------.
|     common object header     |    2 longs
|------------------------------|
| SQLAction instance variables |
|           database           |    2 longs
|           request            |
|------------------------------|
| SQLRequest instance variable |
|         returnedRows         |    1 long
'------------------------------'
```

A first way to define the struct would then be:

```
typedef struct st_SQLAction {
    OBJ_HEADER;
    OOP database;
    OOP request;
    OOP returnedRows;
} *SQLAction;
```

but this results in a lot of duplicated code. Think of what would happen if you had other subclasses of `SQLAction` such as `SQLObjectCreation`, `SQLUpdateQuery`, and so on! The solution, which is also the one used in GNU Smalltalk's source code is to define a macro for each superclass, in this way:

```
/* SQLAction
    |-- SQLRequest
    |     `-- SQLUpdateQuery
    `-- SQLObjectCreation       */

#define ST_SQLACTION_HEADER         \
    OBJ_HEADER;                     \
    OOP database;                   \
    OOP request                     /* no semicolon */

#define ST_SQLREQUEST_HEADER        \
    ST_SQLACTION_HEADER;            \
    OOP returnedRows                /* no semicolon */
```

```
typedef struct st_SQLAction {
    ST_SQLACTION_HEADER;
} *SQLAction;

typedef struct st_SQLRequest {
    ST_SQLREQUEST_HEADER;
} *SQLRequest;

typedef struct st_SQLObjectCreation {
    ST_SQLACTION_HEADER;
    OOP newDBObject;
} *SQLObjectCreation;

typedef struct st_SQLUpdateQuery {
    ST_SQLREQUEST_HEADER;
    OOP numUpdatedRows;
} *SQLUpdateQuery;
```

Note that the macro you declare is used instead of `OBJ_HEADER` in the declaration of both the superclass and the subclasses.

Although this example does not show that, please note that you should not declare anything if the class has indexed instance variables.

The first step in actually using your structs is obtaining a pointer to an OOP which is an instance of your class. Ways to do so include doing a call-in, receiving the object from a call-out (using `#smalltalk`, `#self` or `#selfSmalltalk` as the type specifier).

Let's assume that the `oop` variable contains such an object. Then, you have to dereference the OOP (which, as you might recall from Section 5.4 [Smalltalk types], page 57, point to the actual object only indirectly) and get a pointer to the actual data. You do that with the `OOP_TO_OBJ` macro (note the type casting):

```
SQLAction action = (SQLAction) OOP_TO_OBJ(oop);
```

Now you can use the fields in the object like in this pseudo-code:

```
/* These are retrieved via classNameToOOP and then cached in global
   variables */
OOP sqlUpdateQueryClass, sqlActionClass, sqlObjectCreationClass;
...
invoke_sql_query(
    vmProxy->oopToCObject(action->database),
    vmProxy->oopToString(action->request),
    query_completed_callback,       /* Callback function */
    oop);                           /* Passed to the callback */

...

/* Imagine that invoke_sql_query runs asynchronously and calls this
   when the job is done. */
void
```

```
query_completed_callback(result, database, request, clientData)
     struct query_result *result;
     struct db *database;
     char *request;
     OOP clientData;
{
  SQLUpdateQuery query;
  OOP rows;
  OOP cObject;

  /* Free the memory allocated by oopToString */
  free(request);

  if (OOP_CLASS (oop) == sqlActionClass)
    return;

  if (OOP_CLASS (oop) == sqlObjectCreationClass)
    {
      SQLObjectCreation oc;
      oc = (SQLObjectCreation) OOP_TO_OBJ (clientData);
      cObject = vmProxy->cObjectToOOP (result->dbObject)
      oc->newDBObject = cObject;
    }
  else
    {
      /* SQLRequest or SQLUpdateQuery */
      cObject = vmProxy->cObjectToOOP (result->rows);
      query = (SQLUpdateQuery) OOP_TO_OBJ (clientData);
      query->returnedRows = cObject;
      if (OOP_CLASS (oop) == sqlUpdateQueryClass)
        query->numReturnedRows = vmProxy->intToOOP (result->count);
    }

}
```

Note that the result of `OOP_TO_OBJ` is not valid anymore if a garbage-collection happens; for this reason, you should assume that a pointer to object data is not valid after doing a call-in, calling `objectAlloc`, and using any of the "C to Smalltalk" functions except `intToOOP` (see Section 5.4 [Smalltalk types], page 57). That's why I passed the OOP to the callback, not the object pointer itself.

If your class has indexed instance variables, you can use the `INDEXED_WORD`, `INDEXED_OOP` and `INDEXED_BYTE` macros declared in `gstpub.h`, which return an lvalue for the given indexed instance variable—for more information, see Section 5.7 [Other C functions], page 65.

5.9 Using the Smalltalk environment as an extension library

If you are reading this chapter because you are going to write extensions to GNU Smalltalk, this section won't probably interest you. But if you intend to use GNU Smalltalk as a scripting language or an extension language for your future marvellous software projects, you might be interest.

How to initialize GNU Smalltalk is most briefly and easily explained by looking at GNU Smalltalk's own source code. For this reason, here is a simplified snippet from `gst-tool.c`.

```
int main(argc, argv)
int     argc;
char    **argv;
{
  gst_set_var (GST_VERBOSITY, 1);
  gst_smalltalk_args (argc - 1, argv + 1);
  gst_set_executable_path (argv[0]);
  result = gst_initialize ("kernel-dir", "image-file", GST_NO_TTY);
  if (result != 0)
    exit (result < 0 ? 1 : result);

  if (!gst_process_file ("source-file", GST_DIR_KERNEL_SYSTEM))
    perror ("gst: couldn't load `source-file'");

  gst_invoke_hook (GST_ABOUT_TO_QUIT);
  exit (0);
}
```

Your initialization code will be almost the same as that in GNU Smalltalk's `main()`, with the exception of the call to `gst_process_file`. All you'll have to do is to pass some arguments to the GNU Smalltalk library via `gst_smalltalk_args`, possibly modify some defaults using `gst_get_var` and `gst_set_var`, and then call `gst_initialize`.

Variable indices that can be passed to `gst_get_var` and `gst_set_var` include:

```
GST_DECLARE_TRACING
GST_EXECUTION_TRACING
GST_EXECUTION_TRACING_VERBOSE
GST_GC_MESSAGE
GST_VERBOSITY
GST_MAKE_CORE_FILE
GST_REGRESSION_TESTING
```

While the flags that can be passed as the last parameter to `gst_initialize` are any combination of these:

```
GST_REBUILD_IMAGE
GST_MAYBE_REBUILD_IMAGE
GST_IGNORE_USER_FILES
GST_IGNORE_BAD_IMAGE_NAME
GST_IGNORE_BAD_IMAGE_PATH
GST_IGNORE_BAD_KERNEL_PATH
GST_NO_TTY
```

Note that `gst_initialize` will likely take some time (from a tenth of a second to 3-4 seconds), because it has to check if the image file must be be rebuilt and, if so, it reloads and recompiles the over 50,000 lines of Smalltalk code that form a basic image. To avoid this check, pass a valid image file as the second argument to `gst_initialize`.

The result of `gst_init_smalltalk` is `0` for success, while anything else is an error code.

If you're using GNU Smalltalk as an extension library, you might also want to disable the two `ObjectMemory` class methods, `quit` and `quit:` method. I advice you not to change the Smalltalk kernel code. Instead, in the script that loads your extension classes add these two lines:

```
ObjectMemory class compile: 'quit        self shouldNotImplement'!
ObjectMemory class compile: 'quit: n     self shouldNotImplement'!
```

which will effectively disable the two offending methods. Other possibilities include using `atexit` (from the C library) to exit your program in a less traumatic way, or redefining these two methods to exit through a call out to a C routine in your program.

Also, note that it is not a problem if you develop the class libraries for your programs within GNU Smalltalk's environment (which will not call `defineCFunc` for your own C call-outs), since the addresses of the C call-outs are looked up again when an image is restored.

5.10 Incubator support

The incubator concept provides a mechanism to protect newly created objects from being accidentally garbage collected before they can be attached to some object which is reachable from the root set.

If you are creating some set of objects which will not be immediately (that means, before the next object is allocated from the Smalltalk memory system) be attached to an object which is still "live" (reachable from the root set of objects), you'll need to use this interface.

If you are writing a C call-out from Smalltalk (for example, inside a module), you will not have direct access to the incubator; instead the functions described in Section 5.4 [Smalltalk types], page 57, automatically put the objects that they create in the incubator, and the virtual machine takes care of wrapping C call-outs so that the incubator state is restored at the end of the call.

This section describes its usage from the point of view of a program that is linking with `libgst.a`. Such a program has much finer control to the incubator. The interface provides the following operations:

`void` *INC_ADD_OOP* (*OOP anOOP*) [Macro]

Adds a new object to the protected set.

`inc_ptr` *INC_SAVE_POINTER* () [Macro]
: Retrieves the current incubator pointer. Think of the incubator as a stack, and this operation returns the current stack pointer for later use (restoration) with the incRestorePointer function.

`void` *INC_RESTORE_POINTER* (*inc_ptr ptr*) [Macro]
: Sets (restores) the incubator pointer to the given pointer value.

Typically, when you are within a function which allocates more than one object at a time, either directly or indirectly, you'd want to use the incubator mechanism. First you'd save a copy of the current pointer in a local variable. Then, for each object you allocate (except the last, if you want to be optimal), after you create the object you add it to the incubator's list. When you return, you need to restore the incubator's pointer to the value you got with `INC_SAVE_POINTER` using the `INC_RESTORE_POINTER` macro.

Here's an example from cint.c:

The old code was (the comments are added for this example):

```
desc = (_gst_cfunc_descriptor)
   new_instance_with (cFuncDescriptorClass, numArgs);
desc->cFunction = _gst_cobject_new (funcAddr);    // 1
desc->cFunctionName = _gst_string_new (funcName); // 2
desc->numFixedArgs = FROM_INT (numArgs);
desc->returnType = _gst_classify_type_symbol (returnTypeOOP, true);
for (i = 1; i <= numArgs; i++) {
  desc->argTypes[i - 1] =
   _gst_classify_type_symbol(ARRAY_AT(argsOOP, i), false);
}

return (_gst_alloc_oop(desc));
```

`desc` is originally allocated via `newInstanceWith` and `allocOOP`, two private routines which are encapsulated by the public routine `objectAlloc`. At "1", more storage is allocated, and the garbage collector has the potential to run and free (since no live object is referring to it) desc's storage. At "2" another object is allocated, and again the potential for losing both `desc` and `desc->cFunction` is there if the GC runs (this actually happened!).

To fix this code to use the incubator, modify it like this:

```
OOP     descOOP;
IncPtr  ptr;

incPtr = INC_SAVE_POINTER();
desc = (_gst_cfunc_descriptor)
   new_instance_with (cFuncDescriptorClass, numArgs);
descOOP = _gst_alloc_oop(desc);
INC_ADD_OOP (descOOP);

desc->cFunction = _gst_cobject_new (funcAddr);    // 1
INC_ADD_OOP (desc->cFunction);
```

```
desc->cFunctionName = _gst_string_new (funcName); // 2
/* since none of the rest of the function (or the functions it calls)
 * allocates any storage, we don't have to add desc->cFunctionName
 * to the incubator's set of objects, although we could if we wanted
 * to be completely safe against changes to the implementations of
 * the functions called from this function.
 */

desc->numFixedArgs = FROM_INT (numArgs);
desc->returnType = _gst_classify_type_symbol (returnTypeOOP, true);
for (i = 1; i <= numArgs; i++) {
  desc->argTypes[i - 1] =
   _gst_classify_type_symbol(ARRAY_AT(argsOOP, i), false);
}

return (_gst_alloc_oop(desc));
```

Note that it is permissible for two or more functions to cooperate with their use of the incubator. For example, say function A allocates some objects, then calls function B which allocates some more objects, and then control returns to A where it does some more execution with the allocated objects. If B is only called by A, B can leave the management of the incubator pointer up to A, and just register the objects it allocates with the incubator. When A does a `INC_RESTORE_POINTER`, it automatically clears out the objects that B has registered from the incubator's set of objects as well; the incubator doesn't know about functions A & B, so as far as it is concerned, all of the registered objects were registered from the same function.

6 Tutorial

What this manual presents

This document provides a tutorial introduction to the Smalltalk language in general, and the GNU Smalltalk implementation in particular. It does not provide exhaustive coverage of every feature of the language and its libraries; instead, it attempts to introduce a critical mass of ideas and techniques to get the Smalltalk novice moving in the right direction.

Who this manual is written for

This manual assumes that the reader is acquainted with the basics of computer science, and has reasonable proficiency with a procedural language such as C. It also assumes that the reader is already familiar with the usual janitorial tasks associated with programming: editing, moving files, and so forth.

6.1 Getting started

6.1.1 Starting up Smalltalk

Assuming that GNU Smalltalk has been installed on your system, starting it is as simple as:

```
$ gst
```

the system loads in Smalltalk, and displays a startup banner like:

```
GNU Smalltalk ready

st>
```

You are now ready to try your hand at Smalltalk! By the way, when you're ready to quit, you exit Smalltalk by typing *control-D* on an empty line.

6.1.2 Saying hello

An initial exercise is to make Smalltalk say "hello" to you. Type in the following line (`printNl` is a upper case N and a lower case L):

```
'Hello, world' printNl
```

The system then prints back 'Hello, world' to you. It prints it twice, the first time because you asked to print and the second time because the snipped evaluated to the 'Hello, world' string.[1]

6.1.3 What actually happened

The front-line Smalltalk interpreter gathers all text until a '!' character and executes it. So the actual Smalltalk code executed was:

```
'Hello, world' printNl
```

This code does two things. First, it creates an object of type `String` which contains the characters "Hello, world". Second, it sends the message named `printNl` to the object.

[1] You can also have the system print out a lot of statistics which provide information on the performance of the underlying Smalltalk engine. You can enable them by starting Smalltalk as:

```
$ gst -V
```

When the object is done processing the message, the code is done and we get our prompt back. You'll notice that we didn't say anything about printing ing the string, even though that's in fact what happened. This was very much on purpose: the code we typed in doesn't know anything about printing strings. It knew how to get a string object, and it knew how to send a message to that object. That's the end of the story for the code we wrote.

But for fun, let's take a look at what happened when the string object received the `printNl` message. The string object then went to a table[2] which lists the messages which strings can receive, and what code to execute. It found that there is indeed an entry for `printNl` in that table and ran this code. This code then walked through its characters, printing each of them out to the terminal.[3]

The central point is that an object is entirely self-contained; only the object knew how to print itself out. When we want an object to print out, we ask the object itself to do the printing.

6.1.4 Doing math

A similar piece of code prints numbers:

```
1234 printNl
```

Notice how we used the same message, but have sent it to a new type of object—an integer (from class `Integer`). The way in which an integer is printed is much different from the way a string is printed on the inside, but because we are just sending a message, we do not have to be aware of this. We tell it to `printNl`, and it prints itself out.

As a user of an object, we can thus usually send a particular message and expect basically the same kind of behavior, regardless of object's internal structure (for instance, we have seen that sending `printNl` to an object makes the object print itself). In later chapters we will see a wide range of types of objects. Yet all of them can be printed out the same way—with `printNl`.

White space is ignored, except as it separates words. This example could also have looked like:

```
1234 printNl
```

However, GNU Smalltalk tries to execute each line by itself if possible. If you wanted to write the code on two lines, you might have written something like:

```
(1234
    printNl)
```

From now on, we'll omit `printNl` since GNU Smalltalk does the service of printing the answer for us.

An integer can be sent a number of messages in addition to just printing itself. An important set of messages for integers are the ones which do math:

```
9 + 7
```

[2] Which table? This is determined by the type of the object. An object has a type, known as the class to which it belongs. Each class has a table of methods. For the object we created, it is known as a member of the `String` class. So we go to the table associated with the String class.

[3] Actually, the message `printNl` was inherited from Object. It sent a `print` message, also inherited by Object, which then sent `printOn:` to the object, specifying that it print to the `Transcript` object. The String class then prints its characters to the standard output.

Answers (correctly!) the value 16. The way that it does this, however, is a significant departure from a procedural language.

6.1.5 Math in Smalltalk

In this case, what happened was that the object `9` (an Integer), received a `+` message with an argument of `7` (also an Integer). The `+` message for integers then caused Smalltalk to create a new object `16` and return it as the resultant object. This `16` object was then given the `printNl` message, and printed `16` on the terminal.

Thus, math is not a special case in Smalltalk; it is done, exactly like everything else, by creating objects, and sending them messages. This may seem odd to the Smalltalk novice, but this regularity turns out to be quite a boon: once you've mastered just a few paradigms, all of the language "falls into place". Before you go on to the next chapter, make sure you try math involving `*` (multiplication), `-` (subtraction), and `/` (division) also. These examples should get you started:

```
8 * (4 / 2)
8 - (4 + 1)
5 + 4
2/3 + 7
2 + 3 * 4
2 + (3 * 4)
```

6.2 Using some of the Smalltalk classes

This chapter has examples which need a place to hold the objects they create. Such place is created automatically as necessary; when you want to discard all the objects you stored, write an exclamation mark at the end of the statement.

Now let's create some new objects.

6.2.1 An array in Smalltalk

An array in Smalltalk is similar to an array in any other language, although the syntax may seem peculiar at first. To create an array with room for 20 elements, do[4]:

```
x := Array new: 20
```

The `Array new: 20` creates the array; the `x :=` part connects the name `x` with the object. Until you assign something else to `x`, you can refer to this array by the name `x`. Changing elements of the array is not done using the `:=` operator; this operator is used only to bind names to objects. In fact, you never modify data structures; instead, you send a message to the object, and it will modify itself.

For instance:

```
x at: 1
```

which prints:

```
nil
```

[4] GNU Smalltalk supports completion in the same way as Bash or GDB. To enter the following line, you can for example type '`x := Arr<TAB> new: 20`'. This can come in handy when you have to type long names such as `IdentityDictionary`, which becomes '`Ide<TAB>D<TAB>`'. Everything starting with a capital letter or ending with a colon can be completed.

The slots of an array are initially set to "nothing" (which Smalltalk calls `nil`). Let's set the first slot to the number 99:

```
x at: 1 put: 99
```

and now make sure the 99 is actually there:

```
x at: 1
```

which then prints out:

```
99
```

These examples show how to manipulate an array. They also show the standard way in which messages are passed arguments ments. In most cases, if a message takes an argument, its name will end with ':'.[5]

So when we said `x at: 1` we were sending a message to whatever object was currently bound to `x` with an argument of 1. For an array, this results in the first slot of the array being returned.

The second operation, `x at: 1 put: 99` is a message with two arguments. It tells the array to place the second argument (99) in the slot specified by the first (1). Thus, when we re-examine the first slot, it does indeed now contain 99.

There is a shorthand for describing the messages you send to objects. You just run the message names together. So we would say that our array accepts both the `at:` and `at:put:` messages.

There is quite a bit of sanity checking built into an array. The request

```
6 at: 1
```

fails with an error; 6 is an integer, and can't be indexed. Further,

```
x at: 21
```

fails with an error, because the array we created only has room for 20 objects.

Finally, note that the object stored in an array is just like any other object, so we can do things like:

```
(x at: 1) + 1
```

which (assuming you've been typing in the examples) will print 100.

6.2.2 A set in Smalltalk

We're done with the array we've been using, so we'll assign something new to our `x` variable. Note that we don't need to do anything special about the old array: the fact that nobody is using it any more will be automatically detected, and the memory reclaimed. This is known as *garbage collection* and it is generally done when Smalltalk finds that it is running low on memory. So, to get our new object, simply do:

```
x := Set new
```

which creates an empty set. To view its contents, do:

```
x
```

The kind of object is printed out (i.e., `Set`), and then the members are listed within parenthesis. Since it's empty, we see:

```
Set ()
```

[5] Alert readers will remember that the math examples of the previous chapter deviated from this.

Now let's toss some stuff into it. We'll add the numbers 5 and 7, plus the string 'foo'. This is also the first example where we're using more than one statement, and thus a good place to present the statement separator—the `.` period:

```
x add: 5. x add: 7. x add: 'foo'
```

Like Pascal, and unlike C, statements are separated rather than terminated. Thus you need only use a `.` when you have finished one statement and are starting another. This is why our last statement, `^r`, does not have a `.` following. Once again like Pascal, however, Smalltalk won't complain if your enter a spurious statement separator after *the last* statement.

However, we can save a little typing by using a Smalltalk shorthand:

```
x add: 5; add: 7; add: 'foo'
```

This line does exactly what the previous one did. The trick is that the semicolon operator causes the message to be sent to the same object as the last message sent. So saying `; add: 7` is the same as saying `x add: 7`, because `x` was the last thing a message was sent to.

This may not seem like such a big savings, but compare the ease when your variable is named `aVeryLongVariableName` instead of just `x`! We'll revisit some other occasions where `;` saves you trouble, but for now let's continue with our set. Type either version of the example, and make sure that we've added 5, 7, and "foo":

```
x
```

we'll see that it now contains our data:

```
Set ('foo' 5 7)
```

What if we add something twice? No problem—it just stays in the set. So a set is like a big checklist—either it's in there, or it isn't. To wit:

```
x add:5; add: 5; add: 5; add: 5; yourself
```

We've added *5* several times, but when we printed our set back out, we just see:

```
Set ('foo' 5 7)
```

`yourself` is commonly sent at the end of the cascade, if what you are interested in is the object itself—in this case, we were not interested in the return value of `add: 5`, which happens to be `5` simply. There's nothing magic in `yourself`; it is a unary message like `printNl`, which does nothing but returning the object itself. So you can do this too:

```
x yourself
```

What you put into a set with `add:`, you can take out with `remove:`. Try:

```
x remove: 5
x printNl
```

The set now prints as:

```
Set ('foo' 7)
```

The "5" is indeed gone from the set.

We'll finish up with one more of the many things you can do with a set—checking for membership. Try:

```
x includes: 7
x includes: 5
```

From which we see that x does indeed contain 7, but not 5. Notice that the answer is printed as `true` or `false`. Once again, the thing returned is an object—in this case, an object known as a boolean. We'll look at the use of booleans later, but for now we'll just say that booleans are nothing more than objects which can only either be true or false—nothing else. So they're very useful for answers to yes or no questions, like the ones we just posed. Let's take a look at just one more kind of data structure:

6.2.3 Dictionaries

A dictionary is a special kind of collection. With a regular array, you must index it with integers. With dictionaries, you can index it with any object at all. Dictionaries thus provide a very powerful way of correlating one piece of information to another. Their only downside is that they are somewhat less efficient than simple arrays. Try the following:

```
y := Dictionary new
y at: 'One' put: 1
y at: 'Two' put: 2
y at: 1 put: 'One'
y at: 2 put: 'Two'
```

This fills our dictionary in with some data. The data is actually stored in pairs of key and value (the key is what you give to `at:`—it specifies a slot; the value is what is actually stored at that slot). Notice how we were able to specify not only integers but also strings as both the key and the value. In fact, we can use any kind of object we want as either—the dictionary doesn't care.

Now we can map each key to a value:

```
y at: 1
y at: 'Two'
```

which prints respectively:

```
'One'
2
```

We can also ask a dictionary to print itself:

```
y
```

which prints:

```
Dictionary (1->'One' 2->'Two' 'One'->1 'Two'->2 )
```

where the first member of each pair is the key, and the second the value. It is now time to take a final look at the objects we have created, and send them to oblivion:

```
y
x!
```

The exclamation mark deleted GNU Smalltalk's knowledge of both variables. Asking for them again will return just `nil`.

6.2.4 Closing thoughts

You've seen how Smalltalk provides you with some very powerful data structures. You've also seen how Smalltalk itself uses these same facilities to implement the language. But this is only the tip of the iceberg—Smalltalk is much more than a collection of "neat" facilities

to use. The objects and methods which are automatically available are only the beginning of the foundation on which you build your programs—Smalltalk allows you to add your own objects and methods into the system, and then use them along with everything else. The art of programming in Smalltalk is the art of looking at your problems in terms of objects, using the existing object types to good effect, and enhancing Smalltalk with new types of objects. Now that you've been exposed to the basics of Smalltalk manipulation, we can begin to look at this object-oriented technique of programming.

6.3 The Smalltalk class hierarchy

When programming in Smalltalk, you sometimes need to create new kinds of objects, and define what various messages will do to these objects. In the next chapter we will create some new classes, but first we need to understand how Smalltalk organizes the types and objects it contains. Because this is a pure "concept" chapter, without any actual Smalltalk code to run, we will keep it short and to the point.

6.3.1 Class `Object`

Smalltalk organizes all of its classes as a tree hierarchy. At the very top of this hierarchy is class *Object*. Following somewhere below it are more specific classes, such as the ones we've worked with—strings, integers, arrays, and so forth. They are grouped together based on their similarities; for instance, types of objects which may be compared as greater or less than each other fall under a class known as *Magnitude*.

One of the first tasks when creating a new object is to figure out where within this hierarchy your object falls. Coming up with an answer to this problem is at least as much art as science, and there are no hard-and-fast rules to nail it down. We'll take a look at three kinds of objects to give you a feel for how this organization matters.

6.3.2 Animals

Imagine that we have three kinds of objects, representing *Animals*, *Parrots*, and *Pigs*. Our messages will be *eat*, *sing*, and *snort*. Our first pass at inserting these objects into the Smalltalk hierarchy would organize them like:

Object
- Animals
- Parrots
- Pigs

This means that Animals, Parrots, and Pigs are all direct descendants of *Object*, and are not descendants of each other.

Now we must define how each animal responds to each kind of message.

Animals
- eat –> Say ‘‘I have now eaten”
- sing –> Error
- snort –> Error

Parrots
- eat –> Say ‘‘I have now eaten”
- sing –> Say ‘‘Tweet”
- snort –> Error

```
Pigs
        eat -> Say ‘‘I have now eaten"”
        sing -> Error
        snort -> Say ‘‘Oink”
```

Notice how we kept having to indicate an action for *eat*. An experienced object designer would immediately recognize this as a clue that we haven't set up our hierarchy correctly. Let's try a different organization:

```
Object
    Animals
        Parrots
                Pigs
```

That is, Parrots inherit from Animals, and Pigs from Parrots. Now Parrots inherit all of the actions from Animals, and Pigs from both Parrots and Animals. Because of this inheritance, we may now define a new set of actions which spares us the redundancy of the previous set:

```
Animals
        eat -> Say ‘‘I have now eaten”
        sing -> Error
        snort -> Error
Parrots
        sing -> Say ‘‘Tweet”
Pigs
        snort -> Say ‘‘Oink”
```

Because Parrots and Pigs both inherit from Animals, we have only had to define the *eat* action once. However, we have made one mistake in our class setup—what happens when we tell a Pig to *sing*? It says “Tweet”, because we have put Pigs as an inheritor of Parrots. Let's try one final organization:

```
Object
    Animals
        Parrots
        Pigs
```

Now Parrots and Pigs inherit from Animals, but not from each other. Let's also define one final pithy set of actions:

```
Animals
        eat -> Say ‘‘I have eaten”
Parrots
        sing -> Say ‘‘Tweet”
Pigs
        snort -> Say ‘‘Oink”
```

The change is just to leave out messages which are inappropriate. If Smalltalk detects that a message is not known by an object or any of its ancestors, it will automatically give an error—so you don't have to do this sort of thing yourself. Notice that now sending *sing* to a Pig does indeed not say “Tweet”—it will cause a Smalltalk error instead.

6.3.3 The bottom line of the class hierarchy

The goal of the class hierarchy is to allow you to organize objects into a relationship which allows a particular object to inherit the code of its ancestors. Once you have identified an effective organization of types, you should find that a particular technique need only be implemented once, then inherited by the children below. This keeps your code smaller, and allows you to fix a bug in a particular algorithm in only once place—then have all users of it just inherit the fix.

You will find your decisions for adding objects change as you gain experience. As you become more familiar with the existing set of objects and messages, your selections will increasingly "fit in" with the existing ones. But even a Smalltalk *pro* stops and thinks carefully at this stage, so don't be daunted if your first choices seem difficult and error-prone.

6.4 Creating a new class of objects

With the basic techniques presented in the preceding chapters, we're ready do our first real Smalltalk program. In this chapter we will construct three new types of objects (known as *classes*), using the Smalltalk technique of inheritance to tie the classes together, create new objects belonging to these classes (known as creating instances of the class), and send messages to these objects.

We'll exercise all this by implementing a toy home-finance accounting system. We will keep track of our overall cash, and will have special handling for our checking and savings accounts. From this point on, we will be defining classes which will be used in future chapters. Since you will probably not be running this whole tutorial in one Smalltalk session, it would be nice to save off the state of Smalltalk and resume it without having to retype all the previous examples. To save the current state of GNU Smalltalk, type:

```
ObjectMemory snapshot: 'myimage.im'
```

and from your shell, to later restart Smalltalk from this "snapshot":

```
$ gst -I myimage.im
```

Such a snapshot currently takes a little more than a megabyte, and contains all variables, classes, and definitions you have added.

6.4.1 Creating a new class

Guess how you create a new class? This should be getting monotonous by now—by sending a message to an object. The way we create our first "custom" class is by sending the following message:

```
Object subclass: #Account.
Account instanceVariableNames: 'balance'.
```

Quite a mouthful, isn't it? GNU Smalltalk provides a simpler way to write this, but for now let's stick with this. Conceptually, it isn't really that bad. The Smalltalk variable *Object* is bound to the grand-daddy of all classes on the system. What we're doing here is telling the *Object* class that we want to add to it a subclass known as *Account*. Then, `instanceVariableNames: 'balance'` tells the new class that each of its objects (*instances*) will have a hidden variable named `balance`.

6.4.2 Documenting the class

The next step is to associate a description with the class. You do this by sending a message to the new class:

```
Account comment:
'I represent a place to deposit and withdraw money'
```

A description is associated with every Smalltalk class, and it's considered good form to add a description to each new class you define. To get the description for a given class:

```
Account comment
```

And your string is printed back to you. Try this with class Integer, too:

```
Integer comment
```

However, there is another way to define classes. This still translates to sending objects, but looks more like a traditional programming language or scripting language:

```
Object subclass: Account [
    | balance |
    <comment:
        'I represent a place to deposit and withdraw money'>
]
```

This has created a class. If we want to access it again, for example to modify the comment, we can do so like this:

```
Account extend [
    <comment:
        'I represent a place to withdraw money that has been deposited'>
]
```

This instructs Smalltalk to pick an existing class, rather than trying to create a subclass.

6.4.3 Defining a method for the class

We have created a class, but it isn't ready to do any work for us—we have to define some messages which the class can process first. We'll start at the beginning by defining methods for instance creation:

```
Account class extend [
   new [
       | r |
       <category: 'instance creation'>
       r := super new.
       r init.
       ^r
  ]
]
```

The important points about this are:

- `Account class` means that we are defining messages which are to be sent to the Account class itself.
- `<category: 'instance creation'>` is more documentation support; it says that the methods we are defining supports creating objects of type Account.

- The text starting with `new [` and ending with `]` defined what action to take for the message `new`. When you enter this definition, GNU Smalltalk will simply give you another prompt, but your method has been compiled in and is ready for use. GNU Smalltalk is pretty quiet on successful method definitions—but you'll get plenty of error messages if there's a problem!

  If you're familiar with other Smalltalks, note that the body of the method is always in brackets.

The best way to describe how this method works is to step through it. Imagine we sent a message to the new class Account with the command line:

```
Account new
```

`Account` receives the message `new` and looks up how to process this message. It finds our new definition, and starts running it. The first line, `| r |`, creates a local variable named `r` which can be used as a placeholder for the objects we create. `r` will go away as soon as the message is done being processed; note the parallel with `balance`, which goes away as soon as the object is not used anymore. And note that here you have to declare local variables explicitly, unlike what you did in previous examples.

The first real step is to actually create the object. The line `r := super new` does this using a fancy trick. The word `super` stands for the same object that the message `new` was originally sent to (remember? it's `Account`), except that when Smalltalk goes to search for the methods, it starts one level higher up in the hierarchy than the current level. So for a method in the Account class, this is the Object class (because the class Account inherits from is Object—go back and look at how we created the Account class), and the Object class' methods then execute some code in response to the `#new` message. As it turns out, Object will do the actual creation of the object when sent a `#new` message.

One more time in slow motion: the Account method `#new` wants to do some fiddling about when new objects are created, but he also wants to let his parent do some work with a method of the same name. By saying `r := super new` he is letting his parent create the object, and then he is attaching it to the variable `r`. So after this line of code executes, we have a brand new object of type Account, and `r` is bound to it. You will understand this better as time goes on, but for now scratch your head once, accept it as a recipe, and keep going.

We have the new object, but we haven't set it up correctly. Remember the hidden variable `balance` which we saw in the beginning of this chapter? `super new` gives us the object with the `balance` field containing nothing, but we want our balance field to start at 0.[6]

So what we need to do is ask the object to set itself up. By saying `r init`, we are sending the `init` message to our new Account. We'll define this method in the next section—for now just assume that sending the `init` message will get our Account set up.

Finally, we say `^r`. In English, this is *return what r is attached to.* This means that whoever sent to Account the `new` message will get back this brand new account. At the same time, our temporary variable `r` ceases to exist.

[6] And unlike C, Smalltalk draws a distinction between `0` and `nil`. `nil` is the *nothing* object, and you will receive an error if you try to do, say, math on it. It really does matter that we initialize our instance variable to the number 0 if we wish to do math on it in the future.

6.4.4 Defining an instance method

We need to define the `init` method for our Account objects, so that our `new` method defined above will work. Here's the Smalltalk code:

```
Account extend [
    init [
        <category: 'initialization'>
        balance := 0
    ]
]
```

It looks quite a bit like the previous method definition, except that the first one said `Account class extend`, and ours says `Account extend`.

The difference is that the first one defined a method for messages sent directly to `Account`, but the second one is for messages which are sent to Account objects once they are created.

The method named `init` has only one line, `balance := 0`. This initializes the hidden variable `balance` (actually called an instance variable) to zero, which makes sense for an account balance. Notice that the method doesn't end with `^r` or anything like it: this method doesn't return a value to the message sender. When you do not specify a return value, Smalltalk defaults the return value to the object currently executing. For clarity of programming, you might consider explicitly returning `self` in cases where you intend the return value to be used.[7]

Before going on, ere is how you could have written this code in a single declaration (i.e. without using `extend`):

```
Object subclass: Account [
    | balance |
    <comment:
        'I represent a place to deposit and withdraw money'>
    Account class >> new [
        <category: 'instance creation'>
        | r |
        r := super new.
        r init.
        ^r
    ]
    init [
        <category: 'initialization'>
        balance := 0
    ]
]
```

6.4.5 Looking at our Account

Let's create an instance of class Account:

[7] And why didn't the designers default the return value to nil? Perhaps they didn't appreciate the value of void functions. After all, at the time Smalltalk was being designed, C didn't even have a void data type.

```
a := Account new
```

Can you guess what this does? The `Smalltalk at: #a put: <something>` creates a Smalltalk variable. And the `Account new` creates a new Account, and returns it. So this line creates a Smalltalk variable named `a`, and attaches it to a new Account—all in one line. It also prints the Account object we just created:

```
an Account
```

Hmmm... not very informative. The problem is that we didn't tell our Account how to print itself, so we're just getting the default system `printNl` method—which tells what the object is, but not what it contains. So clearly we must add such a method:

```
Account extend [
    printOn: stream [
        <category: 'printing'>
        super printOn: stream.
        stream nextPutAll: ' with balance: '.
        balance printOn: stream
    ]
]
```

Now give it a try again:

```
a
```

which prints:

```
an Account with balance: 0
```

This may seem a little strange. We added a new method, printOn:, and our printNl message starts behaving differently. It turns out that the printOn: message is the central printing function—once you've defined it, all of the other printing methods end up calling it. Its argument is a place to print to—quite often it is the variable `Transcript`. This variable is usually hooked to your terminal, and thus you get the printout to your screen.

The `super printOn: stream` lets our parent do what it did before—print out what our type is. The `an Account` part of the printout came from this. `stream nextPutAll: ' with balance: '` creates the string `with balance:` , and prints it out to the stream, too; note that we don't use `printOn:` here because that would enclose our string within quotes. Finally, `balance printOn: stream` asks whatever object is hooked to the `balance` variable to print itself to the stream. We set `balance` to 0, so the 0 gets printed out.

6.4.6 Moving money around

We can now create accounts, and look at them. As it stands, though, our balance will always be 0—what a tragedy! Our final methods will let us deposit and spend money. They're very simple:

```
Account extend [
    spend: amount [
        <category: 'moving money'>
        balance := balance - amount
    ]
    deposit: amount [
        <category: 'moving money'>
```

```
            balance := balance + amount
        ]
    ]
```

With these methods you can now deposit and spend amounts of money. Try these operations:

```
a deposit: 125
a deposit: 20
a spend: 10
```

6.4.7 What's next?

We now have a generic concept, an "Account". We can create them, check their balance, and move money in and out of them. They provide a good foundation, but leave out important information that particular types of accounts might want. In the next chapter, we'll take a look at fixing this problem using subclasses.

6.5 Two Subclasses for the Account Class

This chapter continues from the previous chapter in demonstrating how one creates classes and subclasses in Smalltalk. In this chapter we will create two special subclasses of Account, known as Checking and Savings. We will continue to inherit the capabilities of Account, but will tailor the two kinds of objects to better manage particular kinds of accounts.

6.5.1 The Savings class

We create the Savings class as a subclass of Account. It holds money, just like an Account, but has an additional property that we will model: it is paid interest based on its balance. We create the class Savings as a subclass of Account.

```
Account subclass: Savings [
    | interest |
```

This is already telling something: the instance variable `interest` will accumulate interest paid. Thus, in addition to the `spend:` and `deposit:` messages which we inherit from our parent, Account, we will need to define a method to add in interest deposits, and a way to clear the interest variable (which we would do yearly, after we have paid taxes). We first define a method for allocating a new account—we need to make sure that the interest field starts at 0.

We can do so within the `Account subclass: Savings` scope, which we have not closed above.

```
init [
    <category: 'initialization'>
    interest := 0.
    ^super init
]
```

Recall that the parent took care of the `new` message, and created a new object of the appropriate size. After creation, the parent also sent an `init` message to the new object. As a subclass of Account, the new object will receive the `init` message first; it sets up its own instance variable, and then passes the `init` message up the chain to let its parent take care of its part of the initialization.

With our new `Savings` account created, we can define two methods for dealing specially with such an account:

```
interest: amount [
    interest := interest + amount.
    self deposit: amount
]
clearInterest [
    | oldinterest |
    oldinterest := interest.
    interest := 0.
    ^oldinterest
]
```

We are now finished, and close the class scope:

```
]
```

The first method says that we add the `amount` to our running total of interest. The line `self deposit: amount` tells Smalltalk to send ourselves a message, in this case `deposit: amount`. This then causes Smalltalk to look up the method for `deposit:`, which it finds in our parent, Account. Executing this method then updates our overall balance.[8]

One may wonder why we don't just replace this with the simpler `balance := balance + amount`. The answer lies in one of the philosophies of object-oriented languages in general, and Smalltalk in particular. Our goal is to encode a technique for doing something once only, and then re-using that technique when needed. If we had directly encoded `balance := balance + amount` here, there would have been two places that knew how to update the balance from a deposit. This may seem like a useless difference. But consider if later we decided to start counting the number of deposits made. If we had encoded `balance := balance + amount` in each place that needed to update the balance, we would have to hunt each of them down in order to update the count of deposits. By sending `self` the message `deposit:`, we need only update this method once; each sender of this message would then automatically get the correct up-to-date technique for updating the balance.

The second method, `clearInterest`, is simpler. We create a temporary variable `oldinterest` to hold the current amount of interest. We then zero out our interest to start the year afresh. Finally, we return the old interest as our result, so that our year-end accountant can see how much we made.[9]

6.5.2 The Checking class

Our second subclass of Account represents a checking account. We will keep track of two facets:

- What check number we are on
- How many checks we have left in our checkbook

[8] `self` is much like `super`, except that `self` will start looking for a method at the bottom of the type hierarchy for the object, while `super` starts looking one level up from the current level. Thus, using `super` forces inheritance, but `self` will find the first definition of the message which it can.

[9] Of course, in a real accounting system we would never discard such information—we'd probably throw it into a Dictionary object, indexed by the year that we're finishing. The ambitious might want to try their hand at implementing such an enhancement.

We will define this as another subclass of Account:

```
Account subclass: Checking [
    | checknum checksleft |
```

We have two instance variables, but we really only need to initialize one of them—if there are no checks left, the current check number can't matter. Remember, our parent class Account will send us the `init` message. We don't need our own class-specific `new` function, since our parent's will provide everything we need.

```
    init [
        <category: 'initialization'>
        checksleft := 0.
        ^super init
    ]
```

As in Savings, we inherit most of abilities from our superclass, Account. For initialization, we leave `checknum` alone, but set the number of checks in our checkbook to zero. We finish by letting our parent class do its own initialization.

6.5.3 Writing checks

We will finish this chapter by adding a method for spending money through our checkbook. The mechanics of taking a message and updating variables should be familiar:

```
    newChecks: number count: checkcount [
        <category: 'spending'>
        checknum := number.
        checksleft := checkcount
    ]

    writeCheck: amount [
        <category: 'spending'>
        | num |
        num := checknum.
        checknum := checknum + 1.
        checksleft := checksleft - 1.
        self spend: amount.
        ^ num
    ]
]
```

`newChecks:` fills our checkbook with checks. We record what check number we're starting with, and update the count of the number of checks in the checkbook.

`writeCheck:` merely notes the next check number, then bumps up the check number, and down the check count. The message `self spend: amount` resends the message `spend:` to our own object. This causes its method to be looked up by Smalltalk. The method is then found in our parent class, Account, and our balance is then updated to reflect our spending.

You can try the following examples:

```
c := Checking new
c deposit: 250
```

```
c newChecks: 100 count: 50
c writeCheck: 32
c
```

For amusement, you might want to add a printOn: message to the checking class so you can see the checking-specific information.

In this chapter, you have seen how to create subclasses of your own classes. You have added new methods, and inherited methods from the parent classes. These techniques provide the majority of the structure for building solutions to problems. In the following chapters we will be filling in details on further language mechanisms and types, and providing details on how to debug software written in Smalltalk.

6.6 Code blocks

The Account/Saving/Checking example from the last chapter has several deficiencies. It has no record of the checks and their values. Worse, it allows you to write a check when there are no more checks—the Integer value for the number of checks will just calmly go negative! To fix these problems we will need to introduce more sophisticated control structures.

6.6.1 Conditions and decision making

Let's first add some code to keep you from writing too many checks. We will simply update our current method for the Checking class; if you have entered the methods from the previous chapters, the old definition will be overridden by this new one.

```
Checking extend [
    writeCheck: amount [
       | num |

       (checksleft < 1)
           ifTrue: [ ^self error: 'Out of checks' ].
       num := checknum.
       checknum := checknum + 1.
       checksleft := checksleft - 1.
       self spend: amount
       ^ num
   ]
]
```

The two new lines are:

```
(checksleft < 1)
    ifTrue: [ ^self error: 'Out of checks' ].
```

At first glance, this appears to be a completely new structure. But, look again! The only new construct is the square brackets, which appear within a method and not only surround it.

The first line is a simple boolean expression. `checksleft` is our integer, as initialized by our Checking class. It is sent the message `<`, and the argument 1. The current number bound to `checksleft` compares itself against 1, and returns a boolean object telling whether it is less than 1.

Now this boolean, which is either true or false, is sent the message `ifTrue:`, with an argument which is called a code block. A code block is an object, just like any other. But instead of holding a number, or a Set, it holds executable statements. So what does a boolean do with a code block which is an argument to a `ifTrue:` message? It depends on which boolean! If the object is the `true` object, it executes the code block it has been handed. If it is the `false` object, it returns without executing the code block. So the traditional *conditional construct* has been replaced in Smalltalk with boolean objects which execute the indicated code block or not, depending on their truth-value.[10]

In the case of our example, the actual code within the block sends an error message to the current object. `error:` is handled by the parent class Object, and will pop up an appropriate complaint when the user tries to write too many checks. In general, the way you handle a fatal error in Smalltalk is to send an error message to yourself (through the `self` pseudo-variable), and let the error handling mechanisms inherited from the Object class take over.

As you might guess, there is also an `ifFalse:` message which booleans accept. It works exactly like `ifTrue:`, except that the logic has been reversed; a boolean `false` will execute the code block, and a boolean `true` will not.

You should take a little time to play with this method of representing conditionals. You can run your checkbook, but can also invoke the conditional functions directly:

```
true ifTrue: [ 'Hello, world!' printNl ]
false ifTrue: [ 'Hello, world!' printNl ]
true ifFalse: [ 'Hello, world!' printNl ]
false ifFalse: [ 'Hello, world!' printNl ]
```

6.6.2 Iteration and collections

Now that we have some sanity checking in place, it remains for us to keep a log of the checks we write. We will do so by adding a Dictionary object to our Checking class, logging checks into it, and providing some messages for querying our check-writing history. But this enhancement brings up a very interesting question—when we change the "shape" of an object (in this case, by adding our dictionary as a new instance variable to the Checking class), what happens to the existing class, and its objects? The answer is that the old objects are mutated to keep their new shape, and all methods are recompiled so that they work with the new shape. New objects will have exactly the same shape as old ones, but old objects might happen to be initialized incorrectly (since the newly added variables will be simply put to nil). As this can lead to very puzzling behavior, it is usually best to eradicate all of the old objects, and then implement your changes.

If this were more than a toy object accounting system, this would probably entail saving the objects off, converting to the new class, and reading the objects back into the new format. For now, we'll just ignore what's currently there, and define our latest Checking class.

```
Checking extend [
    | history |
```

[10] It is interesting to note that because of the way conditionals are done, conditional constructs are not part of the Smalltalk language, instead they are merely a defined behavior for the Boolean class of objects.

This is the same syntax as the last time we defined a checking account, except that we start with `extend` (since the class is already there). Then, the two instance variables we had defined remain, and we add a new `history` variable; the old methods will be recompiled without errors. We must now feed in our definitions for each of the messages our object can handle, since we are basically defining a new class under an old name.

With our new Checking instance variable, we are all set to start recording our checking history. Our first change will be in the handling of the `init` message:

```
init [
    <category: 'initialization'>
    checksleft := 0.
    history := Dictionary new.
    ^ super init
]
```

This provides us with a Dictionary, and hooks it to our new `history` variable.

Our next method records each check as it's written. The method is a little more involved, as we've added some more sanity checks to the writing of checks.

```
writeCheck: amount [
    <category: 'spending'>
    | num |

    "Sanity check that we have checks left in our checkbook"
    (checksleft < 1)
        ifTrue: [ ^self error: 'Out of checks' ].

    "Make sure we've never used this check number before"
    num := checknum.
    (history includesKey: num)
        ifTrue: [ ^self error: 'Duplicate check number' ].

    "Record the check number and amount"
    history at: num put: amount.

    "Update our next checknumber, checks left, and balance"
    checknum := checknum + 1.
    checksleft := checksleft - 1.
    self spend: amount.
    ^ num
]
```

We have added three things to our latest version of `writeCheck:`. First, since our routine has become somewhat involved, we have added comments. In Smalltalk, single quotes are used for strings; double quotes enclose comments. We have added comments before each section of code.

Second, we have added a sanity check on the check number we propose to use. Dictionary objects respond to the `includesKey:` message with a boolean, depending on whether

something is currently stored under the given key in the dictionary. If the check number is already used, the `error:` message is sent to our object, aborting the operation.

Finally, we add a new entry to the dictionary. We have already seen the `at:put:` message (often found written as `#at:put:`, with a sharp in front of it) at the start of this tutorial. Our use here simply associates a check number with an amount of money spent.[11] With this, we now have a working Checking class, with reasonable sanity checks and per-check information.

Let us finish the chapter by enhancing our ability to get access to all this information. We will start with some simple print-out functions.

```
printOn: stream [
    super printOn: stream.
    ', checks left: ' printOn: stream.
    checksleft printOn: stream.
    ', checks written: ' printOn: stream.
    (history size) printOn: stream.
]
check: num [
    | c |
    c := history
        at: num
        ifAbsent: [ ^self error: 'No such check #' ].
    ^c
]
```

There should be very few surprises here. We format and print our information, while letting our parent classes handle their own share of the work. When looking up a check number, we once again take advantage of the fact that blocks of executable statements are an object; in this case, we are using the `at:ifAbsent:` message supported by the Dictionary class. As you can probably anticipate, if the requested key value is not found in the dictionary, the code block is executed. This allows us to customize our error handling, as the generic error would only tell the user "key not found".

While we can look up a check if we know its number, we have not yet written a way to "riffle through" our collection of checks. The following function loops over the checks, printing them out one per line. Because there is currently only a single numeric value under each key, this might seem wasteful. But we have already considered storing multiple values under each check number, so it is best to leave some room for each item. And, of course, because we are simply sending a printing message to an object, we will not have to come back and re-write this code so long as the object in the dictionary honors our `printNl`/`printOn:` messages sages.

```
printChecks [
    history keysAndValuesDo: [ :key :value |
```

[11] You might start to wonder what one would do if you wished to associate two pieces of information under one key. Say, the value and who the check was written to. There are several ways; the best would probably be to create a new, custom object which contained this information, and then store this object under the check number key in the dictionary. It would also be valid (though probably overkill) to store a dictionary as the value—and then store as many pieces of information as you'd like under each slot!

```
                key print.
                ' - ' print.
                value printNl.
            ]
        ]
    ]
```

We still see a code block object being passed to the dictionary, but `:key :value |` is something new. A code block can optionally receive arguments. In this case, the two arguments represent a key/value pair. If you only wanted the value portion, you could call history with a `do:` message instead; if you only wanted the key portion, you could call history with a `keysDo:` message instead.

We then invoke our printing interface upon them. We don't want a newline until the end, so the `print` message is used instead. It is pretty much the same as `printNl`, since both implicitly use `Transcript`, except it doesn't add a newline.

It is important that you be clear that in principle there is no relationship between the code block and the dictionary you passed it to. The dictionary just invokes the passed code block with a key/value pair when processing a keysAndValuesDo: message. But the same two-parameter code block can be passed to any message that wishes to evaluate it (and passes the exact number of parameters to it). In the next chapter we'll see more on how code blocks are used; we'll also look at how you can invoke code blocks in your own code.

6.7 Code blocks, part two

In the last chapter, we looked at how code blocks could be used to build conditional expressions, and how you could iterate across all entries in a collection.[12] We built our own code blocks, and handed them off for use by system objects. But there is nothing magic about invoking code blocks; your own code will often need to do so. This chapter will shows some examples of loop construction in Smalltalk, and then demonstrate how you invoke code blocks for yourself.

6.7.1 Integer loops

Integer loops are constructed by telling a number to drive the loop. Try this example to count from 1 to 20:

```
1 to: 20 do: [:x | x printNl ]
```

There's also a way to count up by more than one:

```
1 to: 20 by: 2 do: [:x | x printNl ]
```

Finally, counting down is done with a negative step:

```
20 to: 1 by: -1 do: [:x | x printNl ]
```

Note that the `x` variable is local to the block.

```
x
```

just prints `nil`.

[12] The `do:` message is understood by most types of Smalltalk collections. It works for the Dictionary class, as well as sets, arrays, strings, intervals, linked lists, bags, and streams. The `keysDo:` message, for example, works only with dictionaries.

6.7.2 Intervals

It is also possible to represent a range of numbers as a standalone object. This allows you to represent a range of numbers as a single object, which can be passed around the system.

```
i := Interval from: 5 to: 10
i do: [:x | x printNl]
```

As with the integer loops, the Interval class can also represent steps greater than 1. It is done much like it was for our numeric loop above:

```
i := (Interval from: 5 to: 10 by: 2)
i do: [:x| x printNl]
```

6.7.3 Invoking code blocks

Let us revisit the checking example and add a method for scanning only checks over a certain amount. This would allow our user to find "big" checks, by passing in a value below which we will not invoke their function. We will invoke their code block with the check number as an argument ment; they can use our existing check: message to get the amount.

```
Checking extend [
    checksOver: amount do: aBlock
        history keysAndValuesDo: [:key :value |
            (value > amount)
                   ifTrue: [aBlock value: key]
        ]
]
```

The structure of this loop is much like our printChecks message sage from chapter 6. However, in this case we consider each entry, and only invoke the supplied block if the check's value is greater than the specified amount. The line:

```
ifTrue: [aBlock value: key]
```

invokes the user-supplied block, passing as an argument the key, which is the check number. The `value:` message, when received by a code block, causes the code block to execute. Code blocks take `value`, `value:`, `value:value:`, and `value:value:value:` messages, so you can pass from 0 to 3 arguments to a code block.[13]

You might find it puzzling that an association takes a `value` message, and so does a code block. Remember, each object can do its own thing with a message. A code block gets run when it receives a `value` message. An association merely returns the value part of its key/value pair. The fact that both take the same message is, in this case, coincidence.

Let's quickly set up a new checking account with $250 (wouldn't this be nice in real life?) and write a couple checks. Then we'll see if our new method does the job correctly:

```
mycheck := Checking new.
mycheck deposit: 250
mycheck newChecks: 100 count: 40
mycheck writeCheck: 10
mycheck writeCheck: 52
```

[13] There is also a `valueWithArguments:` message which accepts an array holding as many arguments as you would like.

```
mycheck writeCheck: 15
mycheck checksOver: 1 do: [:x | x printNl]
mycheck checksOver: 17 do: [:x | x printNl]
mycheck checksOver: 200 do: [:x | x printNl]
```

We will finish this chapter with an alternative way of writing our `checksOver:` code. In this example, we will use the message `select:` to pick the checks which exceed our value, instead of doing the comparison ourselves. We can then invoke the new resulting collection against the user's code block.

```
Checking extend [
    checksOver: amount do: aBlock [
        | chosen |
        chosen := history select: [:amt| amt > amount].
        chosen keysDo: aBlock
    ]
]
```

Note that `extend` will also overwrite methods. Try the same tests as above, they should yield the same result!

6.8 When Things Go Bad

So far we've been working with examples which work the first time. If you didn't type them in correctly, you probably received a flood of unintelligible complaints. You probably ignored the complaints, and typed the example again.

When developing your own Smalltalk code, however, these messages are the way you find out what went wrong. Because your objects, their methods, the error printout, and your interactive environment are all contained within the same Smalltalk session, you can use these error messages to debug your code using very powerful techniques.

6.8.1 A Simple Error

First, let's take a look at a typical error. Type:

```
7 plus: 1
```

This will print out:

```
7 did not understand selector 'plus:'
<blah blah>
UndefinedObject>>#executeStatements
```

The first line is pretty simple; we sent a message to the `7` object which was not understood; not surprising since the `plus:` operation should have been `+`. Then there are a few lines of gobbledegook: just ignore them, they reflect the fact that the error passed throgh GNU Smalltalk's exception handling system. The remaining line reflect the way the GNU Smalltalk invokes code which we type to our command prompt; it generates a block of code which is invoked via an internal method `executeStatements` defined in class Object and evaluated like `nil executeStatements` (nil is an instance of *UndefinedObject*). Thus, this output tells you that you directly typed a line which sent an invalid message to the `7` object.

All the error output but the first line is actually a stack backtrace. The most recent call is the one nearer the top of the screen. In the next example, we will cause an error which happens deeper within an object.

6.8.2 Nested Calls

Type the following lines:

```
x := Dictionary new
x at: 1
```

The error you receive will look like:

```
Dictionary new: 31 "<0x33788>" error: key not found
...blah blah...
Dictionary>>#error:
[] in Dictionary>>#at:
[] in Dictionary>>#at:ifAbsent:
Dictionary(HashedCollection)>>#findIndex:ifAbsent:
Dictionary>>#at:ifAbsent:
Dictionary>>#at:
UndefinedObject(Object)>>#executeStatements
```

The error itself is pretty clear; we asked for something within the Dictionary which wasn't there. The object which had the error is identified as `Dictionary new: 31`. A Dictionary's default size is 31; thus, this is the object we created with `Dictionary new`.

The stack backtrace shows us the inner structure of how a Dictionary responds to the `#at:` message. Our hand-entered command causes the usual entry for `UndefinedObject(Object)`. Then we see a Dictionary object responding to an `#at:` message (the "Dictionary>>#at:" line). This code called the object with an `#at:ifAbsent:` message. All of a sudden, Dictionary calls that strange method `#findIndex:ifAbsent:`, which evaluates two blocks, and then the error happens.

To understand this better, it is necessary to know that a very common way to handle errors in Smalltalk is to hand down a block of code which will be called when an error occurs. For the Dictionary code, the `at:` message passes in a block of code to the at:ifAbsent: code to be called when `at:ifAbsent:` can't find the given key, and `at:ifAbsent:` does the same with `findIndex:ifAbsent:`. Thus, without even looking at the code for Dictionary itself, we can guess something of the code for Dictionary's implementation:

```
findIndex: key ifAbsent: errCodeBlock [
    ...look for key...
    (keyNotFound) ifTrue: [ ^(errCodeBlock value) ]
    ...
]

at: key [
    ^self at: key ifAbsent: [^self error: 'key not found']
]
```

Actually, `findIndex:ifAbsent:` lies in class `HashedCollection`, as that `Dictionary(HashedCollection)` in the backtrace says.

It would be nice if each entry on the stack backtrace included source line numbers. Unfortunately, at this point GNU Smalltalk doesn't provide this feature. Of course, you have the source code available...

6.8.3 Looking at Objects

When you are chasing an error, it is often helpful to examine the instance variables of your objects. While strategic calls to `printNl` will no doubt help, you can look at an object without having to write all the code yourself. The `inspect` message works on any object, and dumps out the values of each instance variable within the object.[14]

Thus:

```
x := Interval from: 1 to: 5.
x inspect
```

displays:

```
An instance of Interval
start: 1
stop: 5
step: 1
contents: [
    [1]: 1
    [2]: 2
    [3]: 3
    [4]: 4
    [5]: 5
]
```

We'll finish this chapter by emphasizing a technique which has already been covered: the use of the `error:` message in your own objects. As you saw in the case of Dictionary, an object can send itself an `error:` message with a descriptive string to abort execution and dump a stack backtrace. You should plan on using this technique in your own objects. It can be used both for explicit user-caused errors, as well as in internal sanity checks.

6.9 Coexisting in the Class Hierarchy

The early chapters of this tutorial discussed classes in one of two ways. The "toy" classes we developed were rooted at Object; the system-provided classes were treated as immutable entities. While one shouldn't modify the behavior of the standard classes lightly, "plugging in" your own classes in the right place among their system-provided brethren can provide you powerful new classes with very little effort.

This chapter will create two complete classes which enhance the existing Smalltalk hierarchy. The discussion will start with the issue of where to connect our new classes, and then continue onto implementation. Like most programming efforts, the result will leave many possibilities for improvements. The framework, however, should begin to give you an intuition of how to develop your own Smalltalk classes.

[14] When using the Blox GUI, it actually pops up a so-called *Inspector window*.

6.9.1 The Existing Class Hierarchy

To discuss where a new class might go, it is helpful to have a map of the current classes. The following is the basic class hierarchy of GNU Smalltalk. Indentation means that the line inherits from the earlier line with one less level of indentation.[15].

```
Object
  Behavior
    ClassDescription
      Class
      Metaclass
  BlockClosure
  Boolean
    False
    True
  Browser
  CFunctionDescriptor
  CObject
    CAggregate
      CArray
      CPtr
    CCompound
      CStruct
      CUnion
    CScalar
      CChar
      CDouble
      CFloat
      CInt
      CLong
      CShort
      CSmalltalk
      CString
      CUChar
        CByte
          CBoolean
      CUInt
      CULong
      CUShort
  Collection
    Bag
    MappedCollection
    SequenceableCollection
      ArrayedCollection
        Array
        ByteArray
```

[15] This listing is courtesy of the printHierarchy method supplied by GNU Smalltalk author Steve Byrne. It's in the `kernel/Browser.st` file.

```
            WordArray
            LargeArrayedCollection
               LargeArray
               LargeByteArray
               LargeWordArray
            CompiledCode
               CompiledMethod
               CompiledBlock
            Interval
            CharacterArray
               String
                  Symbol
         LinkedList
            Semaphore
         OrderedCollection
            RunArray
            SortedCollection
      HashedCollection
         Dictionary
            IdentityDictionary
               MethodDictionary
            RootNamespace
               Namespace
               SystemDictionary
         Set
            IdentitySet
   ContextPart
      BlockContext
      MethodContext
   CType
      CArrayCType
      CPtrCType
      CScalarCType
   Delay
   DLD
   DumperProxy
      AlternativeObjectProxy
         NullProxy
            VersionableObjectProxy
         PluggableProxy
   File
      Directory
   FileSegment
   Link
      Process
      SymLink
   Magnitude
```

```
  Association
  Character
  Date
  LargeArraySubpart
  Number
    Float
    Fraction
    Integer
      LargeInteger
        LargeNegativeInteger
        LargePositiveInteger
          LargeZeroInteger
      SmallInteger
  Time
Memory
Message
  DirectedMessage
MethodInfo
NullProxy
PackageLoader
Point
ProcessorScheduler
Rectangle
SharedQueue
Signal
  Exception
    Error
      Halt
        ArithmeticError
          ZeroDivide
        MessageNotUnderstood
      UserBreak
    Notification
      Warning
Stream
  ObjectDumper
  PositionableStream
    ReadStream
    WriteStream
      ReadWriteStream
        ByteStream
          FileStream
  Random
  TextCollector
  TokenStream
TrappableEvent
  CoreException
```

```
    ExceptionCollection
  UndefinedObject
  ValueAdaptor
    NullValueHolder
    PluggableAdaptor
      DelayedAdaptor
    ValueHolder
```

While initially a daunting list, you should take the time to hunt down the classes we've examined in this tutorial so far. Notice, for instance, how an Array is a subclass below the *SequenceableCollection* class. This makes sense; you can walk an Array from one end to the other. By contrast, notice how this is not true for Sets: it doesn't make sense to walk a Set from one end to the other.

A little puzzling is the relationship of a Bag to a Set, since a Bag is actually a Set supporting multiple occurrences of its elements. The answer lies in the purpose of both a Set and a Bag. Both hold an unordered collection of objects; but a Bag needs to be optimized for the case when an object has possibly thousands of occurrences, while a Set is optimized for checking object uniqueness. That's why Set being a subclass or Bag, or the other way round, would be a source of problems in the actual implementation of the class. Currently a Bag holds a Dictionary associating each object to each count; it would be feasible however to have Bag as a subclass of HashedCollection and a sibling of Set.

Look at the treatment of numbers—starting with the class *Magnitude*. While numbers can indeed be ordered by *less than*, *greater than*, and so forth, so can a number of other objects. Each subclass of Magnitude is such an object. So we can compare characters with other characters, dates with other dates, and times with other times, as well as numbers with numbers.

Finally, you will have probably noted some pretty strange classes, representing language entities that you might have never thought of as objects themselves: *Namespace*, *Class* and even *CompiledMethod*. They are the base of Smalltalk's "reflection" mechanism which will be discussed later, in Section 6.12.3 [The truth on metaclasses], page 123.

6.9.2 Playing with Arrays

Imagine that you need an array, but alas you need that if an index is out of bounds, it returns nil. You could modify the Smalltalk implementation, but that might break some code in the image, so it is not practical. Why not add a subclass?

```
"We could subclass from Array, but that class is specifically
 optimized by the VM (which assumes, among other things, that
 it does not have any instance variables).  So we use its
 abstract superclass instead.  The discussion below holds
 equally well."
ArrayedCollection subclass: NiledArray [
    <shape: #pointer>

    boundsCheck: index [
        ^(index < 1) | (index > (self basicSize))
    ]
```

```
        at: index [
            ^(self boundsCheck: index)
                ifTrue: [ nil ]
                ifFalse: [ super at: index ]
        ]

        at: index put: val [
            ^(self boundsCheck: index)
                ifTrue: [ val ]
                ifFalse: [ super at: index put: val ]
        ]
    ]
```

Much of the machinery of adding a class should be familiar. We see another declaration like `comment:`, that is `shape:` message. This sets up `NiledArray` to have the same underlying structure of an `Array` object; we'll delay discussing this until the chapter on the nuts and bolts of arrays. In any case, we inherit all of the actual knowledge of how to create arrays, reference them, and so forth. All that we do is intercept `at:` and `at:put:` messages, call our common function to validate the array index, and do something special if the index is not valid. The way that we coded the bounds check bears a little examination.

Making a first cut at coding the bounds check, you might have coded the bounds check in NiledArray's methods twice (once for `at:`, and again for `at:put:`. As always, it's preferable to code things once, and then re-use them. So we instead add a method for bounds checking `boundsCheck:`, and use it for both cases. If we ever wanted to enhance the bounds checking (perhaps emit an error if the index is < 1 and answer nil only for indices greater than the array size?), we only have to change it in one place.

The actual math for calculating whether the bounds have been violated is a little interesting. The first part of the expression returned by the method:

```
(index < 1) | (index > (self basicSize))
```

is true if the index is less than 1, otherwise it's false. This part of the expression thus becomes the boolean object true or false. The boolean object then receives the message `|`, and the argument `(index > (self basicSize))`. `|` means "or"—we want to OR together the two possible out-of-range checks. What is the second part of the expression?[16]

`index` is our argument, an integer; it receives the message `>`, and thus will compare itself to the value `self basicSize` returns. While we haven't covered the underlying structures Smalltalk uses to build arrays, we can briefly say that the `#basicSize` message returns the number of elements the Array object can contain. So the index is checked to see if it's less than 1 (the lowest legal Array index) or greater than the highest allocated slot in the Array. If it is either (the `|` operator!), the expression is true, otherwise false.

From there it's downhill; our boolean object, returned by `boundsCheck:`, receives the `ifTrue:ifFalse:` message, and a code block which will do the appropriate thing. Why

[16] Smalltalk also offers an `or:` message, which is different in a subtle way from `|`. or: takes a code block, and only invokes the code block if it's necessary to determine the value of the expression. This is analogous to the guaranteed C semantic that `||` evaluates left-to-right only as far as needed. We could have written the expressions as `((index < 1) or: [index > (self basicSize)])`. Since we expect both sides of or: to be false most of the time, there isn't much reason to delay evaluation of either side in this case.

do we have `at:put:` return val? Well, because that's what it's supposed to do: look at every implementor of `at:put` or `at:` and you'll find that it returns its second parameter. In general, the result is discarded; but one could write a program which uses it, so we'll write it this way anyway.

6.9.3 Adding a New Kind of Number

If we were programming an application which did a large amount of complex math, we could probably manage it with a number of two-element arrays. But we'd forever be writing in-line code for the math and comparisons; it would be much easier to just implement an object class to support the complex numeric type. Where in the class hierarchy would it be placed?

You've probably already guessed—but let's step down the hierarchy anyway. Everything inherits from Object, so that's a safe starting point. Complex numbers can not be compared with `<` and `>`, and yet we strongly suspect that, since they are numbers, we should place them under the Number class. But Number inherits from Magnitude—how do we resolve this conflict? A subclass can place itself under a superclass which allows some operations the subclass doesn't wish to allow. All that you must do is make sure you intercept these messages and return an error. So we will place our new Complex class under Number, and make sure to disallow comparisons.

One can reasonably ask whether the real and imaginary parts of our complex number will be integer or floating point. In the grand Smalltalk tradition, we'll just leave them as objects, and hope that they respond to numeric messages reasonably. If they don't, the user will doubtless receive errors and be able to track back their mistake with little fuss.

We'll define the four basic math operators, as well as the (illegal) relationals. We'll add `printOn:` so that the printing methods work, and that should give us our Complex class. The class as presented suffers some limitations, which we'll cover later in the chapter.

```
Number subclass: Complex [
    | realpart imagpart |

    "This is a quick way to define class-side methods."
    Complex class >> new [
        <category: 'instance creation'>
        ^self error: 'use real:imaginary:'
    ]
    Complex class >> new: ignore [
        <category: 'instance creation'>
        ^self new
    ]
    Complex class >> real: r imaginary: i [
        <category: 'instance creation'>
        ^(super new) setReal: r setImag: i
    ]

    setReal: r setImag: i [
        <category: 'basic'>
        realpart := r.
```

```
    imagpart := i.
    ^self
]

real [
    <category: 'basic'>
    ^realpart
]
imaginary [
    <category: 'basic'>
    ^imagpart
]

+ val [
    <category: 'math'>
    ^Complex real: (realpart + val real)
        imaginary: (imagpart + val imaginary)
]
- val [
    <category: 'math'>
    ^Complex real: (realpart - val real)
        imaginary: (imagpart - val imaginary)
]
* val [
    <category: 'math'>
    ^Complex real: (realpart * val real) - (imagpart * val imaginary)
        imaginary: (imagpart * val real) + (realpart * val imaginary)
]
/ val [
    <category: 'math'>
    | d r i |
    d := (val real * val real) + (val imaginary * val imaginary).
    r := ((realpart * val real) + (imagpart * val imaginary)).
    i := ((imagpart * val real) - (realpart * val imaginary)).
    ^Complex real: r / d imaginary: i / d
]

= val [
    <category: 'comparison'>
    ^(realpart = val real) & (imagpart = val imaginary)
]

"All other comparison methods are based on <"
< val [
    <category: 'comparison'>
    ^self shouldNotImplement
]
```

```
        printOn: aStream [
            <category: 'printing'>
            realpart printOn: aStream.
            aStream nextPut: $+.
            imagpart printOn: aStream.
            aStream nextPut: $i
        ]
    ]
```

There should be surprisingly little which is actually new in this example. The printing method uses both `printOn:` as well as `nextPut:` to do its printing. While we haven't covered it, it's pretty clear that `$+` generates the ASCII character `+` as an object[17], and `nextPut:` puts its argument as the next thing on the stream.

The math operations all generate a new object, calculating the real and imaginary parts, and invoking the Complex class to create the new object. Our creation code is a little more compact than earlier examples; instead of using a local variable to name the newly-created object, we just use the return value and send a message directly to the new object. Our initialization code explicitly returns self; what would happen if we left this off?

6.9.4 Inheritance and Polymorphism

This is a good time to look at what we've done with the two previous examples at a higher level. With the NiledArray class, we inherited almost all of the functionality ality of arrays, with only a little bit of code added to address our specific needs. While you may have not thought to try it, all the existing methods for an Array continue to work without further effort-you might find it interesting to ponder why the following still works:

```
a := NiledArray new: 10
a at: 5 put: 1234
a do: [:i| i printNl ]
```

The strength of inheritance is that you focus on the incremental changes you make; the things you don't change will generally continue to work.

In the Complex class, the value of polymorphism was exercised. A Complex number responds to exactly the same set of messages as any other number. If you had handed this code to someone, they would know how to do math with Complex numbers without further instruction. Compare this with C, where a complex number package would require the user to first find out if the complex-add function was complex_plus(), or perhaps complex_add(), or add_complex(), or. . .

However, one glaring deficiency is present in the Complex class: what happens if you mix normal numbers with Complex numbers? Currently, the Complex class assumes that it will only interact with other Complex numbers. But this is unrealistic: mathematically, a "normal" number is simply one with an imaginary part of 0. Smalltalk was designed to allow numbers to coerce themselves into a form which will work with other numbers.

The system is clever and requires very little additional code. Unfortunately, it would have tripled the amount of explanation required. If you're interested in how coercion works

[17] A GNU Smalltalk extension allows you to type characters by ASCII code too, as in `$<43>`.

in GNU Smalltalk, you should find the Smalltalk library source, and trace back the execution of the `retry:coercing:` messages. You want to consider the value which the `generality` message returns for each type of number. Finally, you need to examine the `coerce:` handling in each numeric class.

6.10 Smalltalk Streams

Our examples have used a mechanism extensively, even though we haven't discussed it yet. The Stream class provides a framework for a number of data structures, including input and output functionality, queues, and endless sources of dynamically-generated data. A Smalltalk stream is quite similar to the UNIX streams you've used from C. A stream provides a sequential view to an underlying resource; as you read or write elements, the stream position advances until you finally reach the end of the underlying medium. Most streams also allow you to set the current position, providing random access to the medium.

6.10.1 The Output Stream

The examples in this book all work because they write their output to the `Transcript` stream. Each class implements the `printOn:` method, and writes its output to the supplied stream. The `printNl` method all objects use is simply to send the current object a `printOn:` message whose argument is `Transcript` (by default attached to the standard output stream found in the `stdout` global). You can invoke the standard output stream directly:

```
'Hello, world' printOn: stdout
stdout inspect
```

or you can do the same for the Transcript, which is yet another stream:

```
'Hello, world' printOn: stdout
Transcript inspect
```

the last `inspect` statement will show you how the `Transcript` is linked to `stdout`[18].

6.10.2 Your Own Stream

Unlike a pipe you might create in C, the underlying storage of a Stream is under your control. Thus, a Stream can provide an anonymous buffer of data, but it can also provide a stream-like interpretation to an existing array of data. Consider this example:

```
a := Array new: 10
a at: 4 put: 1234
a at: 9 put: 5678
s := ReadWriteStream on: a.
s inspect
s position: 1
s inspect
s nextPut: 11; nextPut: 22
(a at: 1) printNl
a do: [:x| x printNl]
s position: 2
s do: [:x| x printNl]
```

[18] Try executing it under Blox, where the Transcript is linked to the omonymous window!

```
s position: 5
s do: [:x| x printNl]
s inspect
```

The key is the `on:` message; it tells a stream class to create itself in terms of the existing storage. Because of polymorphism, the object specified by on: does not have to be an Array; any object which responds to numeric at: messages can be used. If you happen to have the NiledArray class still loaded from the previous chapter, you might try streaming over that kind of array instead.

You're wondering if you're stuck with having to know how much data will be queued in a Stream at the time you create the stream. If you use the right class of stream, the answer is no. A ReadStream provides read-only access to an existing collection. You will receive an error if you try to write to it. If you try to read off the end of the stream, you will also get an error.

By contrast, WriteStream and ReadWriteStream (used in our example) will tell the underlying collection to grow when you write off the end of the existing collection. Thus, if you want to write several strings, and don't want to add up their lengths yourself:

```
s := ReadWriteStream on: String new
s inspect
s nextPutAll: 'Hello, '
s inspect
s nextPutAll: 'world'
s inspect
s position: 1
s inspect
s do: [:c | stdout nextPut: c ]
s contents
```

In this case, we have used a String as the collection for the Stream. The `printOn:` messages add bytes to the initially empty string. Once we've added the data, you can continue to treat the data as a stream. Alternatively, you can ask the stream to return to you the underlying object. After that, you can use the object (a String, in this example) using its own access methods.

There are many amenities available on a stream object. You can ask if there's more to read with `atEnd`. You can query the position with `position`, and set it with `position:`. You can see what will be read next with `peek`, and you can read the next element with `next`.

In the writing direction, you can write an element with `nextPut:`. You don't need to worry about objects doing a `printOn:` with your stream as a destination; this operation ends up as a sequence of `nextPut:` operations to your stream. If you have a collection of things to write, you can use `nextPutAll:` with the collection as an argument; each member of the collection will be written onto the stream. If you want to write an object to the stream several times, you can use `next:put:`, like this:

```
s := ReadWriteStream on: (Array new: 0)
s next: 4 put: 'Hi!'
s position: 1
s do: [:x | x printNl]
```

6.10.3 Files

Streams can also operate on files. If you wanted to dump the file `/etc/passwd` to your terminal, you could create a stream on the file, and then stream over its contents:

```
f := FileStream open: '/etc/passwd' mode: FileStream read
f linesDo: [ :c | Transcript nextPutAll: c; nl ]
f position: 30
25 timesRepeat: [ Transcript nextPut: (f next) ]
f close
```

and, of course, you can load Smalltalk source code into your image:

```
FileStream fileIn: '/Users/myself/src/source.st'
```

6.10.4 Dynamic Strings

Streams provide a powerful abstraction for a number of data structures. Concepts like current position, writing the next position, and changing the way you view a data structure when convenient combine to let you write compact, powerful code. The last example is taken from the actual Smalltalk source code—it shows a general method for making an object print itself onto a string.

```
printString [
    | stream |
    stream := WriteStream on: (String new).
    self printOn: stream.
    ^stream contents
]
```

This method, residing in Object, is inherited by every class in Smalltalk. The first line creates a WriteStream which stores on a String whose length is currently 0 (`String new` simply creates an empty string. It then invokes the current object with `printOn:`. As the object prints itself to "stream", the String grows to accommodate new characters. When the object is done printing, the method simply returns the underlying string.

As we've written code, the assumption has been that printOn: would go to the terminal. But replacing a stream to a file like `/dev/tty` with a stream to a data structure (`String new`) works just as well. The last line tells the Stream to return its underlying collection, which will be the string which has had all the printing added to it. The result is that the `printString` message returns an object of the String class whose contents are the printed representation of the very object receiving the message.

6.11 Exception handling in Smalltalk

Up to this point of the tutorial, you used the original Smalltalk-80 error signalling mechanism:

```
check: num [
    | c |
    c := history
        at: num
        ifAbsent: [ ^self error: 'No such check #' ].
    ^c
```

```
]
```

In the above code, if a matching check number is found, the method will answer the object associated to it. If no prefix is found, Smalltalk will unwind the stack and print an error message including the message you gave and stack information.

```
CheckingAccount new: 31 "<0x33788>" error: No such check #
...blah blah...
CheckingAccount>>#error:
[] in Dictionary>>#at:ifAbsent:
Dictionary(HashedCollection)>>#findIndex:ifAbsent:
Dictionary>>#at:ifAbsent:
[] in CheckingAccount>>#check:
CheckingAccount>>#check:
UndefinedObject(Object)>>#executeStatements
```

Above we see the object that received the #error: message, the message text itself, and the frames (innermost-first) running when the error was captured by the system. In addition, the rest of the code in methods like `CheckingAccount>>#check:` was not executed.

So simple error reporting gives us most of the features we want:

- Execution stops immediately, preventing programs from continuing as if nothing is wrong.
- The failing code provides a more-or-less useful error message.
- Basic system state information is provided for diagnosis.
- A debugger can drill further into the state, providing information like details of the receivers and arguments on the stack.

However, there is a more powerful and complex error handling mechanism, that is *exception*. They are like "exceptions" in other programming languages, but are more powerful and do not always indicate error conditions. Even though we use the term "signal" often with regard to them, do not confuse them with the signals like `SIGTERM` and `SIGINT` provided by some operating systems; they are a different concept altogether.

Deciding to use exceptions instead of `#error:` is a matter of aesthetics, but you can use a simple rule: use exceptions only if you want to provide callers with a way to recover sensibly from certain errors, and then only for signalling those particular errors.

For example, if you are writing a word processor, you might provide the user with a way to make regions of text read-only. Then, if the user tries to edit the text, the objects that model the read-only text can signal a `ReadOnlyText` or other kind of exception, whereupon the user interface code can stop the exception from unwinding and report the error to the user.

When in doubt about whether exceptions would be useful, err on the side of simplicity; use `#error:` instead. It is much easier to convert an #error: to an explicit exception than to do the opposite.

6.11.1 Creating exceptions

GNU Smalltalk provides a few exceptions, all of which are subclasses of `Exception`. Most of the ones you might want to create yourself are in the `SystemExceptions` namespace. You

can browse the builtin exceptions in the base library reference, and look at their names with `Exception printHierarchy`.

Some useful examples from the system exceptions are `SystemExceptions.InvalidValue`, whose meaning should be obvious, and `SystemExceptions.WrongMessageSent`, which we will demonstrate below.

Let's say that you change one of your classes to no longer support #new for creating new instances. However, because you use the first-class classes feature of Smalltalk, it is not so easy to find and change all sends. Now, you can do something like this:

```
Object subclass: Toaster [
    Toaster class >> new [
        ^SystemExceptions.WrongMessageSent
            signalOn: #new useInstead: #toast:
    ]

    Toaster class >> toast: reason [
        ^super new reason: reason; yourself
    ]

    ...
]
```

Admittedly, this doesn't quite fit the conditions for using exceptions. However, since the exception type is already provided, it is probably easier to use it than #error: when you are doing defensive programming of this sort.

6.11.2 Raising exceptions

Raising an exception is really a two-step process. First, you create the exception object; then, you send it `#signal`.

If you look through the hierarchy, you'll see many class methods that combine these steps for convenience. For example, the class `Exception` provides `#new` and `#signal`, where the latter is just `^self new signal`.

You may be tempted to provide only a signalling variant of your own exception creation methods. However, this creates the problem that your subclasses will not be able to trivially provide new instance creation methods.

```
Error subclass: ReadOnlyText [
    ReadOnlyText class >> signalOn: aText range: anInterval [
        ^self new initText: aText range: anInterval; signal
    ]

    initText: aText range: anInterval [
        <category: 'private'>
        ...
    ]
]
```

Here, if you ever want to subclass `ReadOnlyText` and add new information to the instance before signalling it, you'll have to use the private method `#initText:range:`.

We recommend leaving out the signalling instance-creation variant in new code, as it saves very little work and makes signalling code less clear. Use your own judgement and evaluation of the situation to determine when to include a signalling variant.

6.11.3 Handling exceptions

To handle an exception when it occurs in a particular block of code, use `#on:do:` like this:

```
^[someText add: inputChar beforeIndex: i]
    on: ReadOnlyText
    do: [:sig | sig return: nil]
```

This code will put a handler for `ReadOnlyText` signals on the handler stack while the first block is executing. If such an exception occurs, and it is not handled by any handlers closer to the point of signalling on the stack (known as "inner handlers"), the exception object will pass itself to the handler block given as the `do:` argument.

You will almost always want to use this object to handle the exception somehow. There are six basic handler actions, all sent as messages to the exception object:

`return:` Exit the block that received this `#on:do:`, returning the given value. You can also leave out the argument by sending `#return`, in which case it will be nil. If you want this handler to also handle exceptions in whatever value you might provide, you should use `#retryUsing:` with a block instead.

`retry` Acts sort of like a "goto" by restarting the first block. Obviously, this can lead to an infinite loop if you don't fix the situation that caused the exception.

`#retry` is a good way to implement reinvocation upon recovery, because it does not increase the stack height. For example, this:

```
frobnicate: n [
  ^[do some stuff with n]
      on: SomeError
      do: [:sig | sig return: (self frobnicate: n + 1)]
  ]
```

should be replaced with retry:

```
frobnicate: aNumber [
  | n |
  n := aNumber.
  ^[do some stuff with n]
      on: SomeError
      do: [:sig | n := 1 + n. sig retry]
]
```

`retryUsing:`
Like `#retry`, except that it effectively replaces the original block with the one given as an argument.

`pass` If you want to tell the exception to let an outer handler handle it, use `#pass` instead of `#signal`. This is just like rethrowing a caught exception in other languages.

`resume:` This is the really interesting one. Instead of unwinding the stack, this will effectively answer the argument from the `#signal` send. Code that sends `#signal` to resumable exceptions can use this value, or ignore it, and continue executing. You can also leave out the argument, in which case the `#signal` send will answer nil. Exceptions that want to be resumable must register this capability by answering `true` from the `#isResumable` method, which is checked on every `#resume:` send.

`outer` This is like `#pass`, but if an outer handler uses `#resume:`, this handler block will be resumed (and `#outer` will answer the argument given to `#resume:`) rather than the piece of code that sent `#signal` in the first place.

None of these methods return to the invoking handler block except for `#outer`, and that only in certain cases described for it above.

Exceptions provide several more features; see the methods on the classes `Signal` and `Exception` for the various things you can do with them. Fortunately, the above methods can do what you want in almost all cases.

If you don't use one of these methods or another exception feature to exit your handler, Smalltalk will assume that you meant to `sig return:` whatever you answer from your handler block. We don't recommend relying on this; you should use an explicit `sig return:` instead.

A quick shortcut to handling multiple exception types is the `ExceptionSet`, which allows you to have a single handler for the exceptions of a union of classes:

```
^[do some stuff with n]
    on: SomeError, ReadOnlyError
    do: [:sig | ...]
```

In this code, any `SomeError` or `ReadOnlyError` signals will be handled by the given handler block.

6.11.4 When an exception isn't handled

Every exception chooses one of the above handler actions by default when no handler is found, or they all use `#pass`. This is invoked by sending `#defaultAction` to the class.

One example of a default action is presented above as part of the example of `#error:` usage; that default action prints a message, backtrace, and unwinds the stack all the way.

The easiest way to choose a default action for your own exception classes is to subclass from an exception class that already chose the right one, as explained in the next section. For example, some exceptions, such as warnings, resume by default, and thus should be treated as if they will almost always resume.

Selecting by superclass is by no means a requirement. Specializing your `Error` subclass to be resumable, or even to resume by default, is perfectly acceptable when it makes sense for your design.

6.11.5 Creating new exception classes

If you want code to be able to handle your signalled exceptions, you will probably want to provide a way to pick those kinds out automatically. The easiest way to do this is to subclass `Exception`.

First, you should choose an exception class to specialize. `Error` is the best choice for non-resumable exceptions, and `Notification` or its subclass `Warning` is best for exceptions that should resume with `nil` by default.

Exceptions are just normal objects; include whatever information you think would be useful to handlers. Note that there are two textual description fields, a *description* and a *message text*. The description, if provided, should be a more-or-less constant string answered from a override method on `#description`, meant to describe all instances of your exception class. The message text is meant to be provided at the point of signalling, and should be used for any extra information that code might want to provide. Your signalling code can provide the `messageText` by using `#signal:` instead of `#signal`. This is yet another reason why signalling variants of instance creation messages can be more trouble than they're worth.

6.11.6 Hooking into the stack unwinding

More often useful than even `#on:do:` is `#ensure:`, which guarantees that some code is executed when the stack unwinds, whether because of normal execution or because of a signalled exception.

Here is an example of use of `#ensure:` and a situation where the stack can unwind even without a signal:

```
Object subclass: ExecuteWithBreak [
  | breakBlock |

  break: anObject [
    breakBlock value: anObject
  ]

  valueWithBreak: aBlock [
    "Sets up breakBlock before entering the block,
     and passes self to the block."
    | oldBreakBlock |
    oldBreakBlock := breakBlock.
    ^[breakBlock := [:arg | ^arg].
      aBlock value]
        ensure: [breakBlock := oldBreakBlock]
  ]
]
```

This class provides a way to stop the execution of a block without exiting the whole method as using `^` inside a block would do. The use of `#ensure:` guarantees (hence the name "ensure") that even if `breakBlock` is invoked or an error is handled by unwinding, the old "break block" will be restored.

The definition of `breakBlock` is extremely simply; it is an example of the general unwinding feature of blocks, that you have probably already used:

```
        (history includesKey: num)
            ifTrue: [ ^self error: 'Duplicate check number' ].
```

You have probably been using `#ensure:` without knowing. For example, `File>>#withReadStreamDo:` uses it to ensure that the file is closed when leaving the block.

6.11.7 Handler stack unwinding caveat

One important difference between Smalltalk and other languages is that when a handler is invoked, the stack is not unwound. The Smalltalk exception system is designed this way because it's rare to write code that could break because of this difference, and the `#resume:` feature doesn't make sense if the stack is unwound. It is easy enough to unwind a stack later, and is not so easy to wind it again if done too early.

For almost all applications, this will not matter, but it technically changes the semantics significantly so should be kept in mind. One important case in which it might matter is when using `#ensure:` blocks *and* exception handlers. For comparison, this Smalltalk code:

```
| n |
n := 42.
[[self error: 'error'] ensure: [n := 24]]
    on: Error
    do: [:sig | n printNl. sig return].
n printNl.
```

will put `"42"` followed by `"24"` on the transcript, because the `n := 24` will not be executed until `sig return` is invoked, unwinding the stack. Similar Java code acts differently:

```
int n = 42;
try
  {
    try {throw new Exception ("42");}
    finally {n = 24;}
  }
catch (Exception e)
  {
    System.out.println (n);
  }
System.out.println (n);
```

printing `"24"` twice, because the stack unwinds before executing the catch block.

6.12 Some nice stuff from the Smalltalk innards

Just like with everything else, you'd probably end up asking yourself: how's it done? So here's this chapter, just to wheten your appetite...

6.12.1 How Arrays Work

Smalltalk provides a very adequate selection of predefined classes from which to choose. Eventually, however, you will find the need to code a new basic data structure. Because Smalltalk's most fundamental storage allocation facilities are arrays, it is important that you understand how to use them to gain efficient access to this kind of storage.

The Array Class. Our examples have already shown the Array class, and its use is fairly obvious. For many applications, it will fill all your needs—when you need an array in a new

class, you keep an instance variable, allocate a new Array and assign it to the variable, and then send array accesses via the instance variable.

This technique even works for string-like objects, although it is wasteful of storage. An Array object uses a Smalltalk pointer for each slot in the array; its exact size is transparent to the programmer, but you can generally guess that it'll be roughly the word size of your machine.[19] For storing an array of characters, therefore, an Array works but is inefficient.

Arrays at a Lower Level. So let's step down to a lower level of data structure. A ByteArray is much like an Array, but each slot holds only an integer from 0 to 255-and each slot uses only a byte of storage. If you only needed to store small quantities in each array slot, this would therefore be a much more efficient choice than an Array. As you might guess, this is the type of array which a String uses.

Aha! But when you go back to chapter 9 and look at the Smalltalk hierarchy, you notice that String does not inherit from ByteArray. To see why, we must delve down yet another level, and arrive at the basic methods for setting up the structure of the instances of a class.

When we implemented our NiledArray example, we used `<shape: #inherit>`. The shape is exactly that: the fundamental structure of Smalltalk objects created within a given class. Let's consider the differences in the next sub-sections.

Nothing
: The default shape specifies the simplest Smalltalk object. The object consists only of the storage needed to hold the instance variables. In C, this would be a simple structure with zero or more scalar fields.[20].

`#pointer`
: Storage is still allocated for any instance variables, but the objects of the class must be created with a `new:` message. The number passed as an argument to `new:` causes the new object, in addition to the space for instance variables, to also have that many slots of unnamed (indexed) storage allocated. The analog in C would be to have a dynamically allocated structure with some scalar fields, followed at its end by a array of pointers.

`#byte`
: The storage allocated as specified by new: is an array of bytes. The analog in C would be a dynamically allocated structure with scalar fields[21], followed by a array of `char`.

`#word`
: The storage allocated as specified by new: is an array of C unsigned longs, which are represented in Smalltalk by Integer objects. The analog in C would be a dynamically allocated structure with scalar fields, followed by an array of `long`. This kind of subclass is only used in a few places in Smalltalk.

`#character`
: The storage allocated as specified by new: is an array of characters. The analog in C would be a dynamically allocated structure with scalar fields, followed by a array of `char`.

There are many more shapes for more specialized usage. All of them specify the same kind of "array-like" behavior, with different data types.

---

[19] For GNU Smalltalk, the size of a C `long`, which is usually 32 bits.

[20] C requires one or more; zero is allowed in Smalltalk

[21] This is not always true for other Smalltalk implementations, who don't allow instance variables in variableByteSubclasses and variableWordSubclasses.

How to access this new arrays? You already know how to access instance variables—by name. But there doesn't seem to be a name for this new storage. The way an object accesses it is to send itself array-type messages like `at:`, `at:put:`, and so forth.

The problem is when an object wants to add a new level of interpretation to these messages. Consider a Dictionary—it is a pointer-holding object but its `at:` message is in terms of a key, not an integer index of its storage. Since it has redefined the `at:` message, how does it access its fundamental storage?

The answer is that Smalltalk has defined `basicAt:` and `basicAt:put:`, which will access the basic storage even when the `at:` and `at:put:` messages have been defined to provide a different abstraction.

This can get pretty confusing in the abstract, so let's do an example to show how it's pretty simple in practice. Smalltalk arrays tend to start at 1; let's define an array type whose permissible range is arbitrary.

```
ArrayedCollection subclass: RangedArray [
    | offset |
    <comment: 'I am an Array whose base is arbitrary'>
    RangedArray class >> new: size [
        <category: 'instance creation'>
        ^self new: size base: 1
    ]
    RangedArray class >> new: size base: b [
        <category: 'instance creation'>
        ^(super new: size) init: b
    ]

    init: b [
        <category: 'init'>
        offset := (b - 1).   "- 1 because basicAt: works with a 1 base"
        ^self
   ]
   rangeCheck: i [
        <category: 'basic'>
        (i <= offset) | (i > (offset + self basicSize)) ifTrue: [
            'Bad index value: ' printOn: stderr.
            i printOn: stderr.
            Character nl printOn: stderr.
            ^self error: 'illegal index'
        ]
    ]
    at: [
        self rangeCheck: i.
        ^self basicAt: i - offset
    ]
    at: i put: v [
        self rangeCheck: i.
        ^self basicAt: i - offset put: v
```

```
    ]
]
```

The code has two parts; an initialization, which simply records what index you wish the array to start with, and the at: messages, which adjust the requested index so that the underlying storage receives its 1-based index instead. We've included a range check; its utility will demonstrate itself in a moment:

```
a := RangedArray new: 10 base: 5.
a at: 5 put: 0
a at: 4 put: 1
```

Since 4 is below our base of 5, a range check error occurs. But this check can catch more than just our own misbehavior!

```
a do: [:x| x printNl]
```

Our do: message handling is broken! The stack backtrace pretty much tells the story:

```
RangedArray>>#rangeCheck:
RangedArray>>#at:
RangedArray>>#do:
```

Our code received a do: message. We didn't define one, so we inherited the existing do: handling. We see that an Integer loop was constructed, that a code block was invoked, and that our own at: code was invoked. When we range checked, we trapped an illegal index. Just by coincidence, this version of our range checking code also dumps the index. We see that do: has assumed that all arrays start at 1.

The immediate fix is obvious; we implement our own do:

```
RangedArray extend [
    do: aBlock [
        <category: 'basic'>
        1 to: (self basicSize) do: [:x|
            aBlock value: (self basicAt: x)
        ]
    ]
]
```

But the issues start to run deep. If our parent class believed that it knew enough to assume a starting index of 1[22], why didn't it also assume that it could call basicAt:? The answer is that of the two choices, the designer of the parent class chose the one which was less likely to cause trouble; in fact all standard Smalltalk collections do have indices starting at 1, yet not all of them are implemented so that calling basicAt: would work.[23]

Object-oriented methodology says that one object should be entirely opaque to another. But what sort of privacy should there be between a higher class and its subclasses? How many assumption can a subclass make about its superclass, and how many can the superclass make before it begins infringing on the sovereignty of its subclasses?

Alas, there are rarely easy answers, and this is just an example. For this particular problem, there is an easy solution. When the storage need not be accessed with peak

[22] Actually, in GNU Smalltalk `do:` is not the only message assuming that.

[23] Some of these classes actually redefine `do:` for performance reasons, but they would work even if the parent class' implementation of `do:` was kept.

efficiency, you can use the existing array classes. When every access counts, having the storage be an integral part of your own object allows for the quickest access—but remember that when you move into this area, inheritance and polymorphism become trickier, as each level must coordinate its use of the underlying array with other levels.

6.12.2 Two flavors of equality

As first seen in chapter two, Smalltalk keys its dictionary with things like *#word*, whereas we generally use *'word'*. The former, as it turns out, is from class Symbol. The latter is from class String. What's the real difference between a Symbol and a String? To answer the question, we'll use an analogy from C.

In C, if you have a function for comparing strings, you might try to write it:

```
streq(char *p, char *q)
{
    return (p == q);
}
```

But clearly this is wrong! The reason is that you can have two copies of a string, each with the same contents but each at its own address. A correct string compare must walk its way through the strings and compare each element.

In Smalltalk, exactly the same issue exists, although the details of manipulating storage addresses are hidden. If we have two Smalltalk strings, both with the same contents, we don't necessarily know if they're at the same storage address. In Smalltalk terms, we don't know if they're the same object.

The Smalltalk dictionary is searched frequently. To speed the search, it would be nice to not have to compare the characters of each element, but only compare the address itself. To do this, you need to have a guarantee that all strings with the same contents are the same object. The String class, created like:

```
y := 'Hello'
```

does not satisfy this. Each time you execute this line, you may well get a new object. But a very similar class, Symbol, will always return the same object:

```
y := #Hello
```

In general, you can use strings for almost all your tasks. If you ever get into a performance-critical function which looks up strings, you can switch to Symbol. It takes longer to create a Symbol, and the memory for a Symbol is never freed (since the class has to keep tabs on it indefinitely to guarantee it continues to return the same object). You can use it, but use it with care.

This tutorial has generally used the strcmp()-ish kind of checks for equality. If you ever need to ask the question "is this the same object?", you use the == operator instead of =:

```
x := y := 'Hello'
(x = y) printNl
(x == y) printNl
y := 'Hel', 'lo'
(x = y) printNl
(x == y) printNl
x := #Hello
```

```
y := #Hello
(x = y) printNl
(x == y) printNl
```

Using C terms, `=` compares contents like `strcmp()`. `==` compares storage addresses, like a pointer comparison.

6.12.3 The truth about metaclasses

Everybody, sooner or later, looks for the implementation of the `#new` method in Object class. To their surprise, they don't find it; if they're really smart, they search for implementors of #new in the image and they find out it is implemented by `Behavior`... which turns out to be a subclass of Object! The truth starts showing to their eyes about that sentence that everybody says but few people understand: "classes are objects".

Huh? Classes are objects?!? Let me explain.

Open up an image; then type the text printed in `mono-spaced` font.

```
st> Set superclass!
HashedCollection

st> HashedCollection superclass!
Collection

st> Collection superclass!
Object

st> Object superclass!
nil
```

Nothing new for now. Let's try something else:

```
st> #(1 2 3) class!
Array

st> '123' class!
String

st> Set class!
Set class

st> Set class class!
Metaclass
```

You get it, that strange `Set class` thing is something called "a meta-class"... let's go on:

```
st> ^Set class superclass!
Collection class

st> ^Collection class superclass!
Object class
```

You see, there is a sort of 'parallel' hierarchy between classes and metaclasses. When you create a class, Smalltalk creates a metaclass; and just like a class describes how methods for its instances work, a metaclass describes how class methods for that same class work.

`Set` is an instance of the metaclass, so when you invoke the `#new` class method, you can also say you are invoking an instance method implemented by `Set class`. Simply put, class methods are a lie: they're simply instance methods that are understood by instances of metaclasses.

Now you would expect that `Object class superclass` answers `nil class`, that is `UndefinedObject`. Yet you saw that `#new` is not implemented there... let's try it:

```
st> ^Object class superclass!
Class
```

Uh?!? Try to read it aloud: the `Object class` class inherits from the `Class` class. `Class` is the abstract superclass of all metaclasses, and provides the logic that allows you to create classes in the image. But it is not the termination point:

```
st> ^Class superclass!
ClassDescription

st> ^ClassDescription superclass!
Behavior

st> ^Behavior superclass!
Object
```

Class is a subclass of other classes. `ClassDescription` is abstract; `Behavior` is concrete but lacks the methods and state that allow classes to have named instance variables, class comments and more. Its instances are called *light-weight* classes because they don't have separate metaclasses, instead they all share `Behavior` itself as their metaclass.

Evaluating `Behavior superclass` we have worked our way up to class Object again: Object is the superclass of all instances as well as all metaclasses. This complicated system is extremely powerful, and allows you to do very interesting things that you probably did without thinking about it—for example, using methods such as `#error:` or `#shouldNotImplement` in class methods.

Now, one final question and one final step: what are metaclasses instances of? The question makes sense: if everything has a class, should not metaclasses have one?

Evaluate the following:

```
st> meta := Set class
st> 0 to: 4 do: [ :i |
st>     i timesRepeat: [ Transcript space ]
st>     meta printNl
st>     meta := meta class
st> ]
Set class
 Metaclass
  Metaclass class
   Metaclass
    Metaclass class
```

```
0
```

If you send `#class` repeatedly, it seems that you end up in a loop made of class `Metaclass`[24] and its own metaclass, `Metaclass class`. It looks like class Metaclass is *an instance of an instance of itself.*

To understand the role of `Metaclass`, it can be useful to know that the class creation is implemented there. Think about it.

- `Random class` implements creation and initialization of its instances' random number seed; analogously, `Metaclass class` implements creation and initialization of its instances, which are metaclasses.
- And `Metaclass` implements creation and initialization of its instances, which are classes (subclasses of `Class`).

The circle is closed. In the end, this mechanism implements a clean, elegant and (with some contemplation) understandable facility for self-definition of classes. In other words, it is what allows classes to talk about themselves, posing the foundation for the creation of browsers.

6.12.4 The truth of Smalltalk performance

Everybody says Smalltalk is slow, yet this is not completely true for at least three reasons. First, most of the time in graphical applications is spent waiting for the user to "do something", and most of the time in scripting applications (which GNU Smalltalk is particularly well versed in) is spent in disk I/O; implementing a travelling salesman problem in Smalltalk would indeed be slow, but for most real applications you can indeed exchange performance for Smalltalk's power and development speed.

Second, Smalltalk's automatic memory management is faster than C's manual one. Most C programs are sped up if you relink them with one of the garbage collecting systems available for C or C++.

Third, even though very few Smalltalk virtual machines are as optimized as, say, the Self environment (which reaches half the speed of optimized C!), they do perform some optimizations on Smalltalk code which make them run many times faster than a naive bytecode interpreter. Peter Deutsch, who among other things invented the idea of a just-in-time compiler like those you are used to seeing for Java[25], once observed that implementing a language like Smalltalk efficiently requires the implementor to cheat... but that's okay as long as you don't get caught. That is, as long as you don't break the language semantics. Let's look at some of these optimizations.

For certain frequently used 'special selectors', the compiler emits a send-special-selector bytecode instead of a send-message bytecode. Special selectors have one of three behaviors:

- A few selectors are assigned to special bytecode solely in order to save space. This is the case for `#do:` for example.
- Three selectors (`#at:`, `#at:put:`, `#size`) are assigned to special bytecodes because they are subject to a special caching optimization. These selectors often result in calling a virtual machine primitive, so GNU Smalltalk remembers which primitve was last called

[24] Which turns out to be another subclass of `ClassDescription`.

[25] And like the one that GNU Smalltalk includes as an experimental feature.

as the result of sending them. If we send `#at:` 100 times for the same class, the last 99 sends are directly mapped to the primitive, skipping the method lookup phase.

- For some pairs of receiver classes and special selectors, the interpreter never looks up the method in the class; instead it swiftly executes the same code which is tied to a particular primitive. Of course a special selector whose receiver or argument is not of the right class to make a no-lookup pair is looked up normally.

No-lookup methods do contain a primitive number specification, `<primitive: xx>`, but it is used only when the method is reached through a `#perform:...` message send. Since the method is not normally looked up, deleting the primitive name specification cannot in general prevent this primitive from running. No-lookup pairs are listed below:

`Integer/Integer` `Float/Integer`	for	`+ - * = ~= > < >= <=`
`Float/Float` `Integer/Integer`	for	`// \\ bitOr: bitShift: bitAnd:`
Any pair of objects	for	`== isNil notNil class`
BlockClosure	for	`value value: blockCopy:`[26]

Other messages are open coded by the compiler. That is, there are no message sends for these messages—if the compiler sees blocks without temporaries and with the correct number of arguments at the right places, the compiler unwinds them using jump bytecodes, producing very efficient code. These are:

```
to:by:do: if the second argument is an integer literal
to:do:
timesRepeat:
and:, or:
ifTrue:ifFalse:, ifFalse:ifTrue:, ifTrue:, ifFalse:
whileTrue:, whileFalse:
```

Other minor optimizations are done. Some are done by a peephole optimizer which is ran on the compiled bytecodes. Or, for example, when GNU Smalltalk pushes a boolean value on the stack, it automatically checks whether the following bytecode is a jump (which is a common pattern resulting from most of the open-coded messages above) and combines the execution of the two bytecodes. All these snippets can be optimized this way:

```
1 to: 5 do: [ :i | ... ]
a < b and: [ ... ]
myObject isNil ifTrue: [ ... ]
```

That's all. If you want to know more, look at the virtual machine's source code in `libgst/interp-bc.inl` and at the compiler in `libgst/comp.c`.

6.13 Some final words

The question is always how far to go in one document. At this point, you know how to create classes. You know how to use inheritance, polymorphism, and the basic storage

[26] You won't ever send this message in Smalltalk programs. The compiler uses it when compiling blocks.

management mechanisms of Smalltalk. You've also seen a sampling of Smalltalk's powerful classes. The rest of this chapter simply points out areas for further study; perhaps a newer version of this document might cover these in further chapters.

Viewing the Smalltalk Source Code

Lots of experience can be gained by looking at the source code for system methods; all of them are visible: data structure classes, the innards of the magic that makes classes be themselves objects and have a class, a compiler written in Smalltalk itself, the classes that implement the Smalltalk GUI and those that wrap sockets.

Other Ways to Collect Objects

We've seen Array, ByteArray, Dictionary, Set, and the various streams. You'll want to look at the Bag, OrderedCollection, and SortedCollection classes. For special purposes, you'll want to examine the CObject and CType hierarchies.

Flow of Control

GNU Smalltalk has support for non-preemptive multiple threads of execution. The state is embodied in a Process class object; you'll also want to look at the Semaphore and ProcessorScheduler class.

Smalltalk Virtual Machine

GNU Smalltalk is implemented as a virtual instruction set. By invoking GNU Smalltalk with the `-D` option, you can view the byte opcodes which are generated as files on the command line are loaded. Similarly, running GNU Smalltalk with `-E` will trace the execution of instructions in your methods.

You can look at the GNU Smalltalk source to gain more information on the instruction set. With a few modifications, it is based on the set described in the canonical book from two of the original designers of Smalltalk: *Smalltalk-80: The Language and its Implementation*, by Adele Goldberg and David Robson.

Where to get Help

The Usenet `comp.lang.smalltalk` newsgroup is read by many people with a great deal of Smalltalk experience. There are several commercial Smalltalk implementations; you can buy support for these, though it isn't cheap. For the GNU Smalltalk system in particular, you can try the mailing list at:

```
help-smalltalk@gnu.org
```

No guarantees, but the subscribers will surely do their best!

6.14 A Simple Overview of Smalltalk Syntax

Smalltalk's power comes from its treatment of objects. In this document, we've mostly avoided the issue of syntax by using strictly parenthesized expressions as needed. When this leads to code which is hard to read due to the density of parentheses, a knowledge of Smalltalk's syntax can let you simplify expressions. In general, if it was hard for you to tell how an expression would parse, it will be hard for the next person, too.

The following presentation presents the grammar a couple of related elements at a time. We use an EBNF style of grammar. The form:

```
[ ... ]
```

means that "..." can occur zero or one times.

```
[ ... ]*
```

means zero or more;

```
[ ... ]+
```

means one or more.

```
... | ... [ | ... ]*
```

means that one of the variants must be chosen. Characters in double quotes refer to the literal characters. Most elements may be separated by white space; where this is not legal, the elements are presented without white space between them.

```
methods: ‘‘!’’ id [‘‘class’’] ‘‘methodsFor:’’ string ‘‘!’’ [method ‘‘!’’]+
‘‘!’’
```

Methods are introduced by first naming a class (the id element), specifying "class" if you're adding class methods instead of instance methods, and sending a string argument to the `methodsFor:` message. Each method is terminated with an "!"; two bangs in a row (with a space in the middle) signify the end of the new methods.

```
method: message [pragma] [temps] exprs
message: id | binsel id | [keysel id]+
pragma: ‘‘<’’ keymsg ‘‘>’’
temps: ‘‘|’’ [id]* ‘‘|’’
```

A method definition starts out with a kind of template. The message to be handled is specified with the message names spelled out and identifiers in the place of arguments. A special kind of definition is the pragma; it has not been covered in this tutorial and it provides a way to mark a method specially as well as the interface to the underlying Smalltalk virtual machine. temps is the declaration of local variables. Finally, exprs (covered soon) is the actual code for implementing the method.

```
unit: id | literal | block | arrayconstructor | ‘‘(’’ expr ‘‘)’’
unaryexpr: unit [ id ]+
primary: unit | unaryexpr
```

These are the "building blocks" of Smalltalk expressions. A unit represents a single Smalltalk value, with the highest syntactic precedence. A unaryexpr is simply a unit which receives a number of unary messages. A unaryexpr has the next highest precedence. A primary is simply a convenient left-hand-side name for one of the above.

```
exprs: [expr ‘‘.’’]* [[‘‘^’’] expr]
expr: [id ‘‘:=’’]* expr2

expr2: primary | msgexpr [ ‘‘;’’ cascade ]*
```

A sequence of expressions is separated by dots and can end with a returned value (^). There can be leading assignments; unlike C, assignments apply only to simple variable names. An expression is either a primary (with highest precedence) or a more complex message. cascade does not apply to primary

constructions, as they are too simple to require the construct. Since all primary construct are unary, you can just add more unary messages:

```
1234 printNl printNl printNl
```

`msgexpr: unaryexpr | binexpr | keyexpr`

A complex message is either a unary message (which we have already covered), a binary message (`+`, `-`, and so forth), or a keyword message (`at:`, `new:`, ...) Unary has the highest precedence, followed by binary, and keyword messages have the lowest precedence. Examine the two versions of the following messages. The second have had parentheses added to show the default precedence.

```
myvar at: 2 + 3 put: 4
mybool ifTrue: [ ^ 2 / 4 roundup ]

(myvar at: (2 + 3) put: (4))
(mybool ifTrue: ([ ^ (2 / (4 roundup)) ]))
```

`cascade: id | binmsg | keymsg`

A cascade is used to direct further messages to the same object which was last used. The three types of messages (id is how you send a unary message) can thus be sent.

```
binexpr: primary binmsg [ binmsg ]*
binmsg: binsel primary
binsel: binchar[binchar]
```

A binary message is sent to an object, which primary has identified. Each binary message is a binary selector, constructed from one or two characters, and an argument which is also provided by a primary.

```
1 + 2 - 3 / 4
```

which parses as:

```
(((1 + 2) - 3) / 4)
```

```
keyexpr: keyexpr2 keymsg
keyexpr2: binexpr | primary
keymsg: [keysel keyw2]+
keysel: id‘‘:’’
```

Keyword expressions are much like binary expressions, except that the selectors are made up of identifiers with a colon appended. Where the arguments to a binary function can only be from primary, the arguments to a keyword can be binary expressions or primary ones. This is because keywords have the lowest precedence.

`block: ‘‘[’’ [[‘‘:’’ id]* ‘‘|’’ ] [temps] exprs ‘‘]’’`

A code block is square brackets around a collection of Smalltalk expressions. The leading “: id” part is for block arguments. Note that it is possible for a block to have temporary variables of its own.

`arrayconstructor: ‘‘{’’ exprs ‘‘}’’`

Not covered in this tutorial, this syntax allows to create arrays whose values are not literals, but are instead evaluated at run-time. Compare `#(a b)`, which

results in an Array of two symbols `#a` and `#b`, to `{a. b+c}` which results in an Array whose two elements are the contents of variable `a` and the result of summing `c` to `b`.

```
literal: number | string | charconst | symconst | arrayconst | binding |
eval
arrayconst: ``#'' array | ``#'' bytearray
bytearray: ``['' [number]* ``]''
array: ``('' [literal | array | bytearray | arraysym | ]* ``)''
number: [[dig]+ ``r''] [``-''] [alphanum]+ [``.'' [alphanum]+] [exp
[``-'']dig]+].
string: "'"[char]*"'"
charconst: ``$''char
symconst: ``#''symbol | ``#''string
arraysym: [id | ``:'']*
exp: ``d'' | ``e'' | ``q'' | ``s''
```

We have already shown the use of many of these constants. Although not covered in this tutorial, numbers can have a base specified at their front, and a trailing scientific notation. We have seen examples of character, string, and symbol constants. Array constants are simple enough; they would look like:

```
a := #(1 2 'Hi' $x #Hello 4 16r3F)
```

There are also ByteArray constants, whose elements are constrained to be integers between 0 and 255; they would look like:

```
a := #[1 2 34 16r8F 26r3H 253]
```

Finally, there are three types of floating-point constants with varying precision (the one with the `e` being the less precise, followed by `d` and `q`), and scaled-decimal constants for a special class which does exact computations but truncates comparisons to a given number of decimals. For example, `1.23s4` means "the value `1.23`, with four significant decimal digits".

```
binding: ``#{'' [id ``.'']* id ``}''
```

This syntax has not been used in the tutorial, and results in an Association literal (known as a *variable binding*) tied to the class that is named between braces. For example, `#{Class} value` is the same as `Class`. The dot syntax is required for supporting namespaces: `#{Smalltalk.Class}` is the same as `Smalltalk associationAt: #Class`, but is resolved at compile-time rather than at run-time.

```
symbol: id | binsel | keysel[keysel]*
```

Symbols are mostly used to represent the names of methods. Thus, they can hold simple identifiers, binary selectors, and keyword selectors:

```
#hello
#+
#at:put:
```

```
eval: ``##('' [temps] exprs ``)''
```

This syntax also has not been used in the tutorial, and results in evaluating an arbitrarily complex expression at compile-time, and substituting the result: for

example `##(Object allInstances size)` is the number of instances of `Object` held in the image *at the time the method is compiled.*

```
id: letter[alphanum]*
binchar: ‘‘+’’ | ‘‘-’’ | ‘‘*’’ | ‘‘/’’ | ‘‘~’’ | ‘‘|’’ | ‘‘,’’ |
‘‘<’’ | ‘‘>’’ | ‘‘=’’ | ‘‘&’’ | ‘‘@’’ | ‘‘?’’ | ‘‘\’’ | ‘‘%’’
alphanum: dig | letter
letter: ‘‘A’’..‘‘Z’’
dig: ‘‘0’’..‘‘9’’
```

These are the categories of characters and how they are combined at the most basic level. binchar simply lists the characters which can be combined to name a binary message.

Table of Contents