GNU Smalltalk User's Guide

Version 1.95.10
13 November 2001

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 nice GUI environment including a class browser (see ⟨undefined⟩ [Blox], page ⟨undefined⟩), 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 ⟨undefined⟩ [Class reference], page ⟨undefined⟩. That part of the manual is entirely generated by a Smalltalk program, starting from the source code for the system classes as distributed together with the system.

I'd like to end this introduction reporting an article I posted to Usenet in September 1999. It's about GNU Smalltalk's 'place in the world' and its comparison with another open source Smalltalk, Squeak (a new version of Smalltalk-80 written by lots of people including the great wise behind the original Smalltalk implementation).

Re: GNU Smalltalk and Squeak (was Re: GNU Smalltalk 1.7 development)

I enjoy discussions about the relationship between GNU Smalltalk and Squeak, because I also think about it sometimes. And, believe it or not, I cannot tell which system is more "winning". Actually, they're fundamentally different—the only thing they share is that both are free.

Squeak is Smalltalk for the pure. Smalltalk for *die-hard* object-oriented folks. Don't flame me please, and interpret this article with the necessary irony and benevolence. Why do I say this? Because only a die-hard object-oriented folk, IMO, would write BitBlt and real-time FM synthesis stuff in Smalltalk—even Adele Goldberg wrote her Smalltalk implementation in Smalltalk mostly for clarity (at least she says so).

GNU Smalltalk is Smalltalk for *hacking* object-oriented folks. You know them: they like the object-oriented paradigm because it's clearer and powerful, and maybe they learnt something of design too... but they nevertheless like to toy with C and think that only pointers make them feel like they are "really" programming. Why do I say this? Because I am one of them, and proud of it.

Having said so, I must admit that I have Squeak on my hard disk. And I'd be happy if one of the 'great wise' behind Squeak told me he dignated himself to have a look at my humble creation. Because both do have good points. Let's take a bird's fly on them... Squeak had LargeIntegers ever since it was born, I added them three months ago to a ten-year old GST. Squeak has Namespaces and I already confessed that I'm taking inspiration from them. But GST has closures, it has a better compiler, better C interoperability, and (I beg your pardon Squeakers) an easy to use window framework.

I added closures to GST in three days of work without taking extra caffeine; the Squeak Mailing List has been wondering for two years about when they will be finally there. Having to choose between 1000 downloads a day and Dan Ingalls adding closures to Squeak after peeping at my source code, I'd surely pick the second!!!

Being too loquacious? Probably, so I stop here. The moral is, no civil wars between Smalltalk dialects please; let's all write good programs and let others write good programs. It'll surely be a better world for both.
Paolo Bonzini

1 Installation

1.1 Compiling GNU Smalltalk

The first thing to do to compile GNU Smalltalk is to configure the program, creating the makefiles and a '`gstconf.h`', which contains guesses at the system's peculiarities. This configuration is performed automatically by the '`configure`' shell script; to run it, merely type:

```
./configure
```

Options that you can pass to configure include `--disable-dld`, which precludes Smalltalk programs from dynamically linking libraries at run-time.

After you've configured GNU Smalltalk, you can compile the system by typing:

```
make
```

Smalltalk should compile and link with no errors. If compiling goes wrong you might want to check the commands used to launch the compiler. For example, be sure to check if your compiler has extensions which, if not enabled, don't make it ANSI compatible. If this is the case, type

```
make distclean
CFLAGS=needed command-line flags
```

and retry the configure/make process. In very particular cases, the configure script might miss the presence of a header file or a function on your system. You can patch the '`config.cache`' file created by the configure process. For example, if configure did not find your '`unistd.h`' header file, change the line reading

```
ac_cv_header_unistd_h=${ac_cv_header_unistd_h='no'}
```

to

```
ac_cv_header_unistd_h=${ac_cv_header_unistd_h='yes'}
```

and, again, retry the configure/make process.

The last lines of the make output should be like this:

```
export SMALLTALK_KERNEL=‘cd ./kernel; pwd‘; \
./gst -iQ dummy_file
make[2]: Leaving directory ‘/home/utente/devel-gst’
make[1]: Leaving directory ‘/home/utente/devel-gst’
```

At this point, you have a working GNU Smalltalk. *Congratulations!!!*

You will also want to store the Smalltalk sources and create the image file in the proper place (the image file contains a full snapshot of the status of the system). This is done automatically when you do a `make install`. Specifying `--enable-modules` as an option to configure will load Smalltalk packages in the automatically installed image. For example

```
./configure --enable-modules=Blox,TCP
```

will create an image with the Blox user interface toolkit and the TCP abstraction library built-in.

1.2 Including GNU Smalltalk in your programs (legal information)

Believe it or not, Smalltalk is (also) a very good scripting and macro language. For this reason, GNU Smalltalk is distributed as a library that you can link your programs to. It's very simple: just include '`gstpub.h`', link to '`libgst.a`', and you're done. For more information on what to do in your program so that it communicates well with Smalltalk, relate to ⟨undefined⟩ [C and Smalltalk], page ⟨undefined⟩.

In this case, your program **must** be free, licensed under a license that is compatible with the GNU General Public License, under which the GNU Smalltalk virtual machine is licensed. The Smalltalk programs that we bundle with GNU Smalltalk (currently the browser and the compiler) are also subject to the General Public License, and must be made free.

On the other hand, the core GNU Smalltalk class libraries (standard classes, Blox, TCP abstraction, XML parsing) are released under the GNU Lesser General Public License. It is not necessary (although we would appreciate it) that programs that only use these libraries are made free.

2 Using GNU Smalltalk

2.1 Command line arguments

GNU Smalltalk may be invoked via the following command:

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

When you first invoke GNU Smalltalk, it will attempt to see if any of the kernel method definition files are newer than the last saved binary image in the current directory (if there is one). If there is a newer kernel method definition file, or if the binary image file (called '`gst.im`') does not exist, a new binary image will be built by loading in all the kernel method definition files, performing a full garbage collection in order to compact the space in use, and then saving the resulting data in a position independent format. Your first invocation should look something like this:

```
"Major GC flip... done, used space = 51%"
GNU Smalltalk Ready

st>
```

If you specify `file`, that file will be read and executed and Smalltalk will exit when end of file is reached. If you specify more than one file, each will be read and processed in turn. If you don't specify `file`, standard input is read, and if the standard input is a terminal, a prompt is issued. You may specify '`-`' for the name of a file to invoke an explicit read from standard input; furthermore, specifying '`--`' stops the interpretation of options so that every argument that follows is considered a file name even if it begins with a minus.

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`*

Used to make arguments available to Smalltalk code. The C option parser discards everything after the parameter including -a, while Smalltalk code can get it sending the `arguments` message to the `Smalltalk` object.

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')`

That should be clear.

-c --core-dump
: When this flag is set and a fatal signal occurs, a core dump is produced after an error message is printed. Normally, the backtrace is produced and the system terminates without dumping core.

-d --user-declaration-trace
: Declaration tracing prints the class name, the method name, and the byte codes that the compiler is generating as it compiles methods. Only for files that are named explicitly on the command line; kernel files that are loaded automatically as part of rebuilding the image file do not have their declarations traced.

-D --kernel-declaration-trace
: Like the -d flag, but also includes declarations processed for the kernel files.

-e --user-execution-trace
: Prints the byte codes being executed as the interpreter operates. Only works for those executions that occur after the kernel files have been loaded and the image file dumped.

-E --kernel-declaration-trace
: Like the -e flag, but includes all byte codes executed, whether they occur during the loading of the kernel method definition files, or during the loading and execution of user files.

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

-h -H -? --help
: Prints out a brief summary of the command line syntax of GNU Smalltalk, including the definitions of all of the option flags, and then exits.

-i --rebuild-image
: Ignore the saved image file; always load from the kernel method definition files. Setting this flag bypasses the normal checks for kernel files newer than the image file, or the image file's version stamp out of date with respect to the Smalltalk version. After the kernel definitions have been loaded, a new image file will be saved.

-I file --image-file file
: Use the image file named `file` as the image file to load. Completely bypasses checking the file dates on the kernel files and standard image file.

-l --log-changes
: Produce a log of the compiled Smalltalk code to st-changes.st, in the current working directory.

-L file --log-file file
: Produce a log of the compiled Smalltalk code to the file named `file`.

-q --quiet --silent
: Suppress the printing of execution information while GNU Smalltalk runs. Messages about the beginning of execution or how many byte codes were executed are completely suppressed when this flag is set.

-Q --no-messages
: Suppress the printing of execution information and any other informative message while GNU Smalltalk runs. Useful, for example, for stand-alone shell programs such as CGI scripts.

-r --regression-test
: Disables certain informative I/O; this is used by the regression testing system and is probably not of interest to the general user.

-s --store-no-source
: Usually, GNU Smalltalk stores the methods' source code as FileSegments for files in the kernel directory, and as Strings for files outside it. This behavior minimizes problems on recompile, because FileSegments cannot be relied upon if the source file changes. However, storing source code for big libraries is not useful, since the file-in source is unlikely to change. There are two ways to override this behavior and make GNU Smalltalk store everything loaded in a particular session as FileSegments: one is to specify a directory relative to the kernel directory, such as '`/usr/local/smalltalk/kernel/../blox/Blox.st`'; the other is to specify this option on the command line.

-S --snapshot
: Save a snapshot after loading files from the command line. Of course the snapshot is not saved if you include - (stdin) on the command line and exit by typing Ctrl-C.

-v --version
: Prints out the Smalltalk version number, then exits.

-V --verbose
: Enables verbose mode. When verbose mode is on, various diagnostic messages are printed (currently, only the name of each file as it's loaded).

-y --yacc-debug
: Turns on parser debugging. Not typically used.

2.2 Startup sequence

When GNU Smalltalk is invoked, the first thing it does is choosing two paths, respectively the "image path" and the "kernel path". the image path is set to the value of the `SMALLTALK_IMAGE` environment variable (if it is defined); if `SMALLTALK_IMAGE` is not defined, Smalltalk will try the path compiled in the binary (usually, under Unix systems, '`/usr/local/share/gnu-smalltalk`' or a similar data file path) and then the current directory.

The "kernel path" directory in which to look for each of the kernel method definition files. There are only two possibilities in this case: the directory pointed to by `SMALLTALK_KERNEL` if it is defined, and a subdirectory named '`kernel`' in the current directory. However, kernel files are not required to be in this directory: Smalltalk also knows about a system default location for kernel files, which is compiled in the binary (usually, under Unix systems, '`/usr/local/share/gnu-smalltalk/kernel`' or a similar data file path), and which is used for kernel files not found in the directory chosen as above.

Then, if the `-i` flag is not used, Smalltalk tries to find a saved binary image file in the image path. If this is found, it is checked to be compatible with the current version of Smalltalk and with the current system; Smalltalk is able to load an image created on a system with the same `sizeof(long)` but different endianness (for example, a 68k image on an x86), but not an image created on a system with different `sizeof(long)` like an Alpha image on an x86. Finally, if the images are compatible, it compares the write dates of all of the kernel method definition files against the write date of the binary image file.

If the image is not found, is incompatible, or older than any of the kernel files, a new image has to be created. The set of files that make up the kernel is loaded, one at a time. The list can be found in '`lib/lib.c`', in the `standardFiles` variable. If the image lies in the current directory, or if at least a kernel file was found outside of the system default path, a user-dependant '`.stpre`'[1]

At this point, independent of whether the binary image file was loaded or created, the `initialize` event is sent to the dependants of the special class `ObjectMemory` (see ⟨undefined⟩ [Memory access], page ⟨undefined⟩). After the initialization blocks have been executed, the user initialization file '`.stinit`' is loaded if found in the user's home directory[2].

Finally, if there were any files specified on the command line, they are loaded, otherwise standard input is read and executed until an EOF is detected. You are then able to operate GNU Smalltalk by typing in expressions to the '`st>`' prompt, and/or reading in files that contain Smalltalk code.

At some time, you may wish to abort what GNU Smalltalk is doing and return to the command prompt: you can use `C-c` to do this.

2.3 Syntax of GNU Smalltalk

The language that GNU Smalltalk accepts is based on the *file out* syntax as shown in the *Green Book*, also known as *Smalltalk-80: Bits of History, Words of Advice* by Glenn Krasner. The entire grammar of GNU Smalltalk is described in the '`gst.y`' file, but a brief description may be in order:

```
<statements> !
```

executes the given statements immediately. For example,

```
16rFFFF printNl !
```

prints out the decimal value of hex `FFFF`, followed by a newline.

```
Smalltalk quitPrimitive !
```

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

```
! class expression methodsFor: category name !
method definition 1 !
method definition 2 !
```

---

[1] The file is called '`_stpre`' under MS-DOS and '`.gstpre`' on the Atari ST. Under OSes that don't use home directories it is looked for in the current directory.

[2] The same considerations made above hold here too. The file is called '`_stinit`' under MS-DOS and '`.gstinit`' on the Atari ST, and is looked for in the current directory under OSes that don't use home directories.

```
...
method definition n ! !
```

This syntax is used to define new methods in a given class. 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. Two consecutive bangs terminate the set of method definitions. `category name` should be a string object that describes what category to file the methods in.

```
!Float methodsFor: 'pi calculations'!

radiusToArea
    ^self squared * Float pi !

radiusToCircumference
    ^self * 2 * Float pi ! !
```

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 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.

The return operator, which is represented in the Blue Book as an up-arrow, is mapped to the ASCII caret symbol `^`.

A complete treatment of the Smalltalk syntax and of the class library can be found in the included tutorial and class reference. More information on the implementation of the language can be found in the *Blue Book*; the relevant parts can are also available online as HTML documents, at `http://users.ipa.net/~dwighth/smalltalk/bluebook/bluebook_imp_toc.html`.

2.4 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. The only system that I know of which currently fails the test suite is the NeXT, and this is apparently due to their non-standard C runtime libraries.

The test suite is by no means exhaustive. See ⟨undefined⟩ [Future], page ⟨undefined⟩.

3 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).

3.1 Memory accessing methods

GNU Smalltalk provides methods for directly accessing real memory. You may access memory either as individual bytes, or as 32 bit words. You may read the contents of memory, or write to it. You may also find out the size and alignment of scalar C types, and 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.

bigEndian Method on `Memory`

Answers `true` on machine architectures where the most significant byte of a 32 bit integer has the lowest address (e.g. 68000 and Sparc), and false on architectures where the least significant byte occurs at the lowest address (e.g. Intel and VAX).

C*Size Variable

C*Alignment Variable

The `CIntSize`, `CLongSize`, `CShortSize`, `CFloatSize`, `CDoubleSize`, `CPtrSize`, `CDoubleAlignment` globals are provided by the VM as part of the Smalltalk dictionary, and are there for compatibility with old versions of GNU Smalltalk. However you should not use them and, instead, send messages like `CInt sizeof` or `CDouble alignof`.

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.). For examples of using `asOop` and `asObject`, look at the Blox source code in '`blox/tk/BloxBasic.st`'.

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 dependants 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.

3.2 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.]*

3.2.1 Introduction

The standard Smalltalk-80 programming environment 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). Most important global variables are classes defining the behavior of objects.

In a development dealing with modelling of real systems, polymorphic symbolic identification is often needed. This means that it should be possible to use the same name for different classes or other global variables. Let us mention class Module as an example which would mean totaly different things for a programmer, for a computer technician and for a civil engineer or an architect.

This issue becomes inevitable if we start to work in a Smalltalk environment supporting persistence. Polymorphism of classes becomes necessary in the moment we start to think about storing classes in the database since after restoring them into the running Smalltalk image a mismatch with the current symbolic identification of the present classes could

occur. For example you might have the class Module already in your image with the meaning of a program module (e.g. in a CASE system) and you might attempt to load the class Module representing a part of the computer hardware from the database for hardware configuration system. The last class could get bound to the #Module symbol in the Smalltalk system dictionary and the other class could remain in the system as unbound class with full functionality, however, it could not be accessed anymore at the symbolical level in the source code.

Objects which have to be identified in the source code of methods or message sends by their names are included in Smalltalk which is a sole instance of SystemDictionary. Such objects may be identified simply by stating their name as primary in a Smalltalk statement. The code is compiled in the Smalitalk environment and if such a primary is found it is bound to the corresponding object as receiver of the rest of the message send. In this way Smalltalk as instance of SystemDictionary represents the sole symbolic name space in the Smalltalk system. In the following text the symbolic name space will be called simply environment to make the text more clear.

3.2.2 Concepts

To support polymorphic symbolical identification several environments will be needed. The same name may be located concurrently in several environments and point to diverse objects.

However, symbolic navigation between these environments is needed. Before approaching the problem of the syntax to be implemented and of its very implementation, we have to point out which structural relations are going to be established between environments.

Since the environment has first to be symbolically identified to gain access to its global variables, it has to be a global variable in another environment. Obviously, `Smalltalk` will be the first environment from which the navigation begins. From `Smalltalk` some of the existing environments may be seen. From these environments other sub-environments may be seen, etc. This means that environments represent nodes in a graph where symbolic identifications from one environment to another one represent branches.

However, 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. unability to use recursive algorithms. This is why in general the environments build a directed acyclic graph[1].

Let us call the partial ordering relation which occurs between the two environments to be inheritance. Sub-environments inherits from their super-environments.

Not only that "inheritance" is the standard term for the partial ordering relation in the lattice theory but the feature of inheritance in the meaning of object-orientation is associated with this relation. This means that 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

[1] An inheritance tree in the current GNU Smalltalk implementation of namespaces; a class can fake multiple inheritance by specifying a namespace (environment, if you prefer) as one of its pool dictionaries.

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 from Smalltalk all global variables defined in it, it is not necessary to define a special global variable pointing to root 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 and lookups via methods such as `#at:` and `#includesKey:`.

3.2.3 Syntax

Global objects of an environment (local or inherited) may be referenced by their symbol 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 has to be referenced from another environment (i.e. which is not on the inheritance link) 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 Smalltalk through the tree of sub-environments).

For the identification of global objects in another environment a "pathname" of symbols is used. The symbols are separated by blanks, i.e. the "look" to be implemented is that of

```
Smalltalk Tasks MyTask
```

and of

```
Super Super Peter.
```

Its similarity to a sequence of message sends is not casual, and suggests the following syntax for write access:[2]

```
Smalltalk Tasks MyTask: anotherTask
```

This resembles the way accessors are used for other objects. As it is custom in Smalltalk, however, we are reminded by uppercase letters that we are accessing global objects.

In addition, a special syntax has been implemented that returns the Association object for a particular global: so the last example above can be written also like

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

This special kind of literal (called a *variable binding*) is also valid inside literal arrays.

3.2.4 Implementation

A superclass of `SystemDictionary` called `RootNamespace` has to be defined and many of the features of Smalltalk-80 SystemDictionaries will be hosted by that class. `Namespace` and `SystemDictionary` will in turn become subclasses of `RootNamespace`. One could wonder why is RootNamespace superior to Namespace in the hierarchy: the answer is that it is more convenient because root namespaces are more similar to Dictionaries, not having to handle the burden of lookup into super-environments.

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

[2] Absent from the original paper.

Accessors like `#at:ifAbsent:` and `#includesKey:`
: Inheritance has to be implemented.

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

For programs to be able to process correctly the "pathnames" and the accessors, this feature must be implemented directly in RootNamespace; it is easily handled through the standard `doesNotUnderstand:` message for trapping message sends that the virtual machine could not resolve. `RootNamespace` 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 the structured environments which is first of all a framework for class polymorphism.

In Smalltalk the classes have the instance variable `name` which holds the name of the class. Each defined class is included in Smalltalk 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 variable of Class which has to be defined and properly set in relevant methods. In the mother environment the class should be included under its name.

Of course, any class (just like any other object) may be included concurrently in several environments, even under different symbols in the same or in diverse environments. We can consider this 'alias names' of the particular class or global variable. However, classes may be referenced under the other names or in other environments as their mother environment e.g. for the purpose of intance creation or messages to he class (class methods), but they cannot be compiled in other environment. If a class compiles its methods it always compiles them in its mother environment even if this compilation is requested from another environment. If the syntax is not correct in the mother environment, a compilation error simply occurs.

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 has to be change to reflect the environment in which it is shown, i.e. the method '`nameIn: environment`' has to be implemented and used on proper places.

These methods are not all which have to redefined in the Smalltalk system to achieve full functionality of structured environments. In particular, changes have to be made to the behavior classes, to the user interface, to the compiler, to a few classes supporting persistance. An interesting point that could not be noticed is that the environment is easier to use if evaluations (*doits*) are parsed as if UndefinedObject's mother environment was *the current namespace*.

3.2.5 Using namespaces

Using namespaces if often merely a matter of rewriting the loading script this way:

```
Smalltalk addSubspace: #NewNS!
Namespace current: NewNS!
...
```

```
Namespace current: Smalltalk!
```

Also remember that pool dictionaries are actually "pool namespaces", in the sense that including a namespace in the pool dictionaries list will automatically include its superspaces too. Declaring a namespace as a pool dictionaries is similar in this way to C++'s `using namespace` declaration.

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

```
Smalltalk Set variableSubclass: #Set
    ...
    category: 'My application-Extensions'
```

or the equivalent syntax `Set extend`, 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 Arrays, Strings, etc. too), primitive operations will still answer standard Smalltalk SmallIntegers, and so on. Or, variableWordSubclasses will recognize 32-bit `Smalltalk LargeInteger` objects, but not LargeIntegers belonging to your own namespace.

Unfortunately this problem is not easy to solve since Smalltalk has to cache 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 and pool dictionary names, so you might want to give them a try. An example of using namespaces is given by the '`examples/Publish.st`' file in the GNU Smalltalk source code directory.

3.3 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 asynchronous I/O: 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. For more information on these classes, look in the class reference.

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 Blox GUI, instead, the Transcript Window will be attached to the Transcript object (see ⟨undefined⟩ [Blox], page ⟨undefined⟩).

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.

3.4 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.

3.5 Special kinds of object

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

- finalizable objects
- weak objects (i.e. objects whose contents are not considered, during garbage collection, when scanning the heap for live objects).
- read-only objects (like literals found in methods)
- 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 GC 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.

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 `COobject`s that would be stale when the image is loaded again, causing a segmentation violation as soon as they are accessed by the finalization method.

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
- WeakKeyLookupTable
- WeakValueLookupTable
- WeakIdentitySet
- WeakKeyIdentityDictionary
- WeakValueIdentityDictionary

3.6 The context unwinding system

When the C code files in something, the doits in the file-in are evaluated in a different execution environment so that the file-in is protected from errors signalled in the doit; the same also happens, for example, when you do a C callin to Smalltalk.

Now, suppose that you want to evaluate something using Behavior's `evaluate:` method. Instead of using complex exception handling code, the unwinding system can be used to obtain that behavior with code as simple as this:

```
[ lastResult := evalFor perform: selector ] valueWithUnwind.
```

In fact, using `valueWithUnwind` arranges things so that an exception will resume execution after the block, instead of stopping it.

This system is quite low-level, so it should not be used in most cases: as a rule of thumb, use it only when the corresponding C code uses the `prepareExecutionEnvironment`, `stopExecuting` and `finishExecutionEnvironment` functions. The places where I'm using it obey this rule. They include the exception handling system, the ST parser, `Behavior>>#evalString:to:` and `Behavior>>#evalString:to:ifError:`.

The `valueWithUnwind` method is defined in '`BlkClosure.st`', but it mostly relies on code in the '`ContextPart.st`' file. Here are the methods involved in the unwinding mechanism: you can find descriptions in the Smalltalk source code. Note that all these methods are internal, this reference is provided just for sake of completeness.

- ContextPart class>>#unwind
- ContextPart class>>#unwind:
- ContextPart>>#mark
- ContextPart>>#returnTo:
- Process>>#unwindPoints

3.7 Packages

Thanks to Andreas Klimas' insight, GNU Smalltalk now includes a powerful packaging system which allows one to file in components (*goodies* in Smalltalk's very folkloristic terminology) without caring of whether they need other goodies to be loaded.

The packaging system is implemented by a Smalltalk class, `PackageLoader`, which looks for information about packages in the file named (guess what) '`packages`', in the current image directory. 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: #('Blox' 'Browser').
PackageLoader fileInPackage: 'Compiler'.
```

The second way is to use the '`Load.st`' file which lies in the GST image directory. For example, start GNU Smalltalk with this command line:

```
gst -qK Load.st -a Browser Blox Compiler
```

[3]

and GST will automatically file in:

[3] When using an alternate image path, don't use the -K option and pass the full path to the '`Load.st`' script.

- BloxTK, needed by Blox
- Blox, loaded first because Browser needs it
- Tokenizer, not specified on the command line, but needed by Parser
- Parser, also not specified, but needed by Browser and Compiler
- Browser
- Compiler (Blox is skipped because it has already been loaded)

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

To provide support for this system, you have to give away with your GST goodies a small file (say you call it '`mypkg`') which looks like this:

```
# This is the comment for the package file for
# the absolutely WONDERFUL package MyPackage

MyPackage
    MyPrereq1 MyPrereq2 C:MyCallout1 C:MyCallout2
    MyFilein1.st MyFilein2.st libmymodule
    ../yourDirectoryName      # absolute or relative to kernel path
```

then people who install your package will only have to do

```
gst-package mypkg
```

which is a small shell script which will execute these two commands

```
cat packages mypkg > packages
gst -qK Load.st -a MyPackage
```

Simple, isn't it? For examples of package declarations, have a look at the '`packages`' file as distributed with GNU Smalltalk.

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

3.7.1 Blox

Blox is a GUI building block tool kit. It is an abstraction on top of the a platform's native GUI toolkit that is common across all platforms. Writing to the Blox interface means your GUI based application will be portable to any platform where Blox is supported.

Blox is a wrapper around other toolkits, which constitutes the required portability layer; currently the only one supported is Tcl/Tk but alternative versions of Blox, for example based on Gtk+ and GNOME, will be considered. Instead of having to rewrite widgets and support for each platform, Blox simply asks the other toolkit to do so (currently, it hands valid Tcl code to a standard Tcl 8.0 environment); the abstraction from the operating system being used is then extracted out of GNU Smalltalk.

Together with the toolkit, the '`blox`' directory contains a browsing system that 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, some simple debugging tools are provided. An Inspector window allows the programmer to graphically inspect and modify the representation of an object and, based on Steve Byrne's original Blox-based browser, a walkback inspector was designed which will display a backtrace when the program encounters an error. Finally, the

Transcript global object is redirected to print to the transcript window instead of printing to stdout.

This browser evolved from a Motif-based version developed around 1993 written by Brad Diller (bdiller@docent.com). Because of legal concerns about possible copyright infringement because his initial implementation used parts of ParcPlace's Model-View-Controller (MVC) message interface, he and Richard Stallman devised a new window update scheme which is more flexible and powerful than MVC's dependency mechanism, and allowed him to purge all the MVC elements from the implementation.

Four years later I—Paolo Bonzini—further improved the code to employ a better class design (for example, Brad used Dictionaries for classes still to be fleshed out) and be aesthetically more appealing (taking advantage of the new parser and Blox text widget, I added syntax highlighting to the code browsers).

To start the browser you can simply type:

```
gst -qK blox/Run.st
```

This will load any requested packages, then, if all goes well, a worksheet window with a menu named *Smalltalk* will appear in the top-left corner of the screen. You might want to file-in '`blox/Run.st`' from your '`.stinit`' file (see ⟨undefined⟩ [Startup sequence], page ⟨undefined⟩) or to run it automatically through ObjectMemory (see ⟨undefined⟩ [Memory access], page ⟨undefined⟩).

3.7.2 The Smalltalk-in-Smalltalk compiler

The Smalltalk-in-Smalltalk compiler is a nice compiler for Smalltalk code which is written in Smalltalk itself. Ideally, the C compiler would only serve to bootstrap the system, then a fully working Smalltalk compiler would start compiling methods.

The current status of the Smalltalk-in-Smalltalk compiler can be summarized thus: it does work, but it does not work well. This for many reasons: first of all it is slow (10-15 times slower than the C compiler), and it does not produce very optimized code. Anyway it has very few bugs (it does have some), it is a good example of programming the GNU Smalltalk system, and its source code (found in the '`compiler`' directory) provides good insights into the Smalltalk virtual machine: so, after all, it is not that bad. If want to give it a try, just file in the Compiler package.

The compiler is built on a recursive descent parser which creates parse nodes in the form of instances of subclasses of `STParseNode`. Then the parser instantiates a compiler object which creates the actual method; more information on the inner workings of the compiler can be found in the comment for the `STCompiler` class.

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

- First, actions are not hard-coded in the parser itself: the parser creates a parse tree, then hands it to methods in `STParser` that can be overridden in different `STParser` subclasses. This is done by the compiler itself, in which a subclass of `STParser` (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 `STFormatter`.

- Third, just like all recursive descent parsers, it is pretty easy to figure out which part of the stream a method takes care of parsing, and you can override the parsing methods in a subclass to do "something interesting" even while a parse tree is created. This is demonstrated by the syntax highlighting engine included with the browser, implemented by the `STPluggableParser` and `BCode` classes.

3.7.3 Dynamic loading through the DLD package

DLD is the Dynamic LoaDer package. This is a peculiar package in that it is always loaded if your system supports it; currently supported architectures include `dlopen` (used in Linux, BSD, Solaris and many more systems), Win32, HPUX, GNU DLD (Linux ELF), AIX and GNU libtool (which includes a portable `dlopen` for a lot of systems).

The DLD package 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: '/usr/lib/libc.a'
```

You will then be able to use `#defineCFunc:...` (see ⟨undefined⟩ [C callout], page ⟨undefined⟩) 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 DLD when using GNU Smalltalk as an extension language. To disable DLD, configure GNU Smalltalk passing the `--without-dld` switch.

Note that a DLD class will be present even if DLD is disabled (either because your system is not supported, or by the `--without-dld` configure switch) but any attempt to perform dynamic linking will result in an error.

3.7.4 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 the `I18N` package to ease you in doing so.

Internationalizing software 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 "Hey, 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.[4]

The package provides much more functionality, including more advanced formatting options support for Unicode, and conversion to and from several character sets (including ISO-8859, KOI-8, and East-Asian double-byte character sets). For more information, refer to the sources and to the class 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

[4] 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.

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.

3.7.5 The SUnit testing package

`SUnit` is a framework to write and perform test cases in Smalltalk, originarily written by the father of Extreme Programming[5], 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`.

3.7.5.1 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

3.7.5.2 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.

[5] 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.

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.

3.7.5.3 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:

```
SetTestCasee>>#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.

3.7.5.4 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.

3.7.6 TCP, WebServer, NetworkSupport

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 time-out checking 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.

In addition, package `WebServer` implements a servlet-based web serving framework engine, including support for file servers as well as Wiki-style servers[6]; each server is a subclass of `Servlet`, and different servers can live together under different paths. See the class side examples protocol of WebServer to get it up and running quick.

The server is based on the GPL'ed WikiWorks project. For up to date/more info go see `http://wiki.cs.uiuc.edu/VisualWorks/WikiWorks>`. Many thanks go to the various people who had worked on the version on which the server is based:

Joseph Bacanskas joeb@mutual.navigant.com
Travis Griggs tgriggs@keyww.com
Ralph Johnson johnson@cs.uiuc.edu
Eliot Miranda eliot@objectshare.com
Ken Treis ktreis@keyww.com
John Brant brant@cs.uiuc.edu
Joe Whitesell whitesell@physsoft.com

Apart from porting to GNU Smalltalk, a number of changes were made to the code, including refactoring of classes, better aesthetics, authentication support, and HTTP 1.1 compliance.

There is also code implementing the most popular Internet protocols: FTP, HTTP, NNTP, SMTP, POP3 and IMAP. These classes are derived from multiple public domain and open-source packages available for other Smalltalk dialect and ported to GNU Smalltalk.

[6] A Wiki is a kind of collaborative web site, which allows one to edit the contents of a page.

3.7.7 An XML parser and object model for GNU Smalltalk

The XML parser library for Smalltalk, loaded as package VWXML (XML is still reserved for the InDelv parser provided with old versions of GNU Smalltalk) includes a validating XML parser and Document Object Model. In future versions the InDelv parser will be removed and this library will be loaded when the PackageLoader is asked for the XML package. Unluckily, the libraries are incompatible because they come from two completely different sources.

There were many reasons to upgrade to the VisualWorks library. First of all, it is rapidly becoming a standard in the Smalltalk world; it looks more like Smalltalk than the InDelv parser, which is written in a minimal Smalltalk subset so that its source can be automatically converted to Java; it is a validating parser and in general more modern (for example it supports XML namespaces); and finally, an XSL interpreter based on it is available as open-source and will be ported to GNU Smalltalk soon.

The parser's classes are loaded in their own namespace, named XML.

Documentation for the parser is not available yet. To have some clue, look at the class-side protocol for `XML XMLParser` and at the '`printing`' protocol for `XML Node` and its subclasses.

3.7.8 Minor packages

Various other "minor" packages are provided, typically as examples of writing modules for GNU Smalltalk (see ⟨undefined⟩ [External modules], page ⟨undefined⟩). These are Regex, providing Perl5 regular expressions, GDBM, which is an interface to the GNU database manager, and MD5, which provides a simple class to quickly compute cryptographically strong hash values.

4 Interoperability between C and GNU Smalltalk

4.1 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 ⟨undefined⟩ [Packages], page ⟨undefined⟩), and to fit in with the GNU Smalltalk packaging system, is likely to be a five-minutes task.

If your goodie is mostly written in C and you don't 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; the method to be used here is `DLD class>>#addLibrary:`.

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

Modules are searched for in the '`gnu-smalltalk`' subdirectory of the system library path, or in the directory that the `SMALLTALK_MODULES` environment variable points to. 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 Blox:

```
void
gst_initModule(proxy)
     VMProxy *proxy;
{
  vmProxy = proxy;
  vmProxy->defineCFunc("Tcl_Eval", Tcl_Eval);
  vmProxy->defineCFunc("Tcl_GetStringResult", Tcl_GetStringResult);
  vmProxy->defineCFunc("tclInit", tclInit);
  vmProxy->defineCFunc("bloxIdle", bloxIdle);
}
```

Note that the `defineCFunc` function is called through a function pointer in `gst_initModule`, and that Blox saves the value of its parameter to be used 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 functions directly, but through a `VMProxy` exported by '`libgst.a`' and accessible through the `interpreterProxy` variable.

First of all, you have to build your package as a shared library; using GNU Automake and `libtool`, this is as easy as changing your '`Makefile.am`' file so that it reads like this

[1] The most notable are AIX and Windows.

```
pkglib_LTLIBRARIES = libblox.la
libblox_la_LDFLAGS = -module -no-undefined ... more flags ...
libblox_la_SOURCES = ... your source files ...
```

instead of reading like this

```
pkglib_LIBRARIES = libblox.a
libblox_a_LDFLAGS = ... more flags ...
libblox_a_SOURCES = ... your source files ...
```

As you see, you only have to change '`.a`' extensions to '`.la`', `LIBRARIES` targets to `LTLIBRARIES`, and add appropriate options to `LDFLAGS`[2]. You will also have to run `libtoolize` and follow its instruction, but this is really simpler than it looks.

Note that this example uses '`pkglib`' because TCP is installed together with Smalltalk, but in general this is not necessary. You can install the library wherever you want; `libtool` will even generate appropriate warnings to the installer if `ldconfig` (or an equivalent program) has to be re-run.

Finally, you will have to specify the name of the module in the '`packages`' file. In this case, the relevant entry in that file will be

```
Blox
    Kernel
    Blox.st libblox
    ../blox
```

Everything not ending with '`.st`' will be picked by the package loader as a module, and will be passed to `DLD class>>#addModule:` before attempting to file-in the Smalltalk source files.

4.2 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 '`cint.c`'. In the function `initCFuncs`, the following code appears:

```
extern int system();
extern char *getenv();

defineCFunc("system", system);
defineCFunc("getenv", getenv);
```

Any functions that you will call from Smalltalk must be similarly defined.

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 automatically generated

[2] Specifying `-no-undefined` is not necessary, but it does perform that the portability conditions explained above (no reference to symbols in the virtual machine) are satisfied

by calling a method which is available to every class, `defineCFunc:withSelectorArgs:returning:args:`. The method that was used in old versions of GNU Smalltalk, `defineCFunc:withSelectorArgs:forClass:returning:args:`, is still present for backward compatibility, but its use is deprecated and should be avoided.

Here are the definitions for the two functions `system` and `getenv` (taken from '`CFuncs.st`')

```
SystemDictionary defineCFunc: 'system'
    withSelectorArgs: 'system: aString'
    returning: #int
    args: #(#string)!

SystemDictionary defineCFunc: 'getenv'
    withSelectorArgs: 'getenv: aString'
    returning: #string
    args: #(#string)!
```

The various keyword arguments are described below.

The arguments are as follows:

`SystemDictionary`
: This specifies where the new method should be stored. In our case, the method will be installed in the SystemDictionary, so that we would invoke it thus:

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

Again, 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 sbb@gnu.org' !
```

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

`withSelectorArgs: 'system: aString'`
: This defines how this method will be invoked from Smalltalk. 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 legal 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
`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 Float

`void` No returned value

`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 ⟨undefined⟩ [Smalltalk types], page ⟨undefined⟩).

ctype You can pass an instance of CType or one of its subclasses (see ⟨undefined⟩ [C data types], page ⟨undefined⟩)

`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`

`string` passed as `char *`

`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

`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`

`cObject` C object value passed as `long` or `void *`

`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 *
char	Boolean (True, False)	int
char	Character	int (C promotion rule)
char	Integer	int
double	Float	double (C promotion)
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

4.3 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 `CmumbleCType`. 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 `CmumbleType` (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 `Cmumble`. These subclasses can answer size and alignment information.

Instances of `CObject` holds a pointer to a C type variable. The variable have been allocated from Smalltalk by doing `type new`, where `type` is a `CType` subclass instance, or it may have been returned through the C callout mechanism as a return value. Thinking about this facet of the implementation (that `CObject` point to C objects) tends to confuse me when I'm thinking about having CObjects which are, say, of type `long*`... so I try to think of CObject as just representing a C data object and not thinking about the implementation. To talk about the *type* `long*`, you'd create an instance of CPtrCType (because all `CType` instances represent C types, not C objects), via

```
"use the existing CLongCType instance"
CPtrCType elementType: CLongType.
```

To allocate one of these C objects, you'd do:

```
longPtr := (CPtrCType elementType: CLongType) new.
```

Now you have a C variable of type "long *" accessible from longPtr.

Scalars fetch their value when sent the `value` message, and change their value when sent the `value:` message.

CStrings can be indexed using `at:` with a zero based index, which returns a Smalltalk `Character` instance corresponding to the indexed element of the string. To change the value at a given index, use `at:put:`.

To produce a pointer to a character, use `addressAt:`. To dereference the string, like `*(char *)foo`, use `deref`: this returns an object of type `CChar`, not a `Character` instance). To replace the first character in the string, use `deref:` and pass in a `CChar` instance. These operations aren't real useful for `CStrings`, but they are present for completeness and for

symmetry with pointers: after all, you can say `*string` in C and get the first character of the string, just like you can say `*string = 'f'`.

Also for symmetry (but this is useful in fact) `+ anInteger` returns a `CString` object pointing to `integer` bytes from the start of the string. `-` acts like `+` if it is given an integer as its parameter. If a pointer 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.

`replaceWith: aString` replaces the string the instance points to with the new string. Actually, it copies the bytes from the Smalltalk String instance aString into the C string object, and null terminates. Be sure that the C string has enough room! You can also use a Smalltalk ByteArray as the data source.

Instances of `CArray` represent an array of some C data. The underlying element type is provided by a `CType` subclass instance which is associated with the `CPtr` instance. They have `at:` and `at:put:` operations just like Strings. `at:` returns a Smalltalk datatype for the given element of the array (if the element type is a scalar, otherwise it returns a `CObject` subclass instance whose type is that of the element type); `at:put:` works similarly. `addressAt:` returns a CObject subclass instance no matter what, which you then can send `value` or or `value:` to get or set its value. CArray's also support `deref`, `deref:`, `+` and `-` with equivalent semantics to CString.

CPtrs are similar to CArrays (as you might expect given the similarity between pointers and arrays in C) and even more similar to CStrings (as you might again expect since strings are pointers in C). In fact both CPtrs and CArrays are subclasses of a common subclass, CAggregate. Just like CArrays, the underlying element type is provided by a `CType` subclass instance which is associated with the `CPtr` instance.

CPtr's also have `value` and `value:` which get or change the underlying value that's pointed to. Like CStrings, they have #incr, #decr, #incrBy: and #decrBy:. They also have #+ and #- which do what you'd expect.

Finally, there are `CStruct` and `CUnion`, which are abstract subclasses of `CObject`[3]. In the following I 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)
- `char` (unsigned too) & byte type
- `double` (and `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:

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

```
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))
        classVariableNames: ''
        poolDictionaries: ''
        category: 'C interface-Audio'
!

CStruct subclass: #AudioInfo
        declaration: #( (#play #{AudioPrinfo} )
                        (#record #{AudioPrinfo} )
                        (#monitorGain #uLong)
```

```
                        (#yyy (#array #uLong 4)))
            classVariableNames: ''
            poolDictionaries: ''
            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 `instanceVariableNames` replaced by `declaration`[4]. 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 '`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`
`#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))
```

[4] The old `#newStruct:declaration:` method for creating CStructs is deprecated because it does not allow one to set the category.

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.

4.4 Manipulating Smalltalk data from C

GNU Smalltalk internally maps every object except Integers to a data structure named an *OOP* (which is not an acronym for anything, as far as I know). 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 ⟨undefined⟩ [Linking your libraries to the virtual machine], page ⟨undefined⟩. 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 Float and returns the C `double` for that integer.

`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.

`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.

`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.

`voidPtr` **OOPToCObject** (`OOP`) Function

This functions assumes that the passed OOP is a kind of CObject and returns a C `voidPtr` 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 `voidPtr` 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 `voidPtr`.

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 ⟨undefined⟩ [Incubator], page ⟨undefined⟩). 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`. Note that Smalltalk Integers are always signed and have a bit less of precision with respect to C longs. On 32 bit machines, their precision is 30 bits (if unsigned) or 31 bits (if signed); on 64 bit machines, their precision is 62 bits (if unsigned) or 63 bits (if signed).

`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 Float which contains the same value as the passed `double`. Unlike Integers, Floats have exactly the same precision as C 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` **classNameToOOP** (`char *`) Function

This method returns the Smalltalk class (i.e. an instance of a subclass of Class) whose name is the given parameter. 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` **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** (`voidPtr`) 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** (`voidPtr, 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 ⟨undefined⟩ [Smalltalk callin], page ⟨undefined⟩), 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:

`void` **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

Remove an occurrence of the given OOP from the registry.

4.5 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).

`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 = msgSend(
    intToOOP(1),
    symbolToOOP("+"),
    intToOOP(2),
    nil);
```

`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 = strMsgSend(
    intToOOP(1),
    "+",
    intToOOP(2),
    nil);
```

`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] = nil;
/* ... some more code here ... */

shouldBeThreeOOP = 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 = nvmsgSend(
    intToOOP(1),
    symbolToOOP("+"),
    &argument,
    1);
```

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;
msgSend(&shouldBeThree, "%i %i + %i", 1, 2)

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

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

/* 'This is a test', ' ok?' */
char *str;
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 nil, 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 ⟨undefined⟩ [Smalltalk types], page ⟨undefined⟩). *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)
b	int	True or False (see boolToOOP)

c	char	Character (see charToOOP)
C	voidPtr	CObject (see cObjToOOP)
s	char *	String (see stringToOOP)
S	char *	Symbol (see symbolToOOP)
o	OOP	any
t	char *, voidPtr	CObject (see below)
T	OOP, voidPtr	CObject (see below)

'%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 ⟨undefined⟩ [Smalltalk types], page ⟨undefined⟩); instead, if you specify '%T', you will have to directly pass an OOP for the new CObject's type.

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 an Float
b	0	int	nil or a Boolean
c	'\0'	char	nil or a Character
C	NULL	voidPtr	nil or a CObject
s	NULL	char *	nil, a String, or a Symbol
?	0	char *, voidPtr	See oopToC
o	nilOOP	OOP	any (result is not converted)
v		/	any (result is discarded)

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

4.6 Other functions available to modules

In addition to the functions above, the VMProxy that is made 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 'gstpub.h'.

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.

Caution: This is the only function in the intepreterProxy that can be called from within a signal handler.

void syncWait (OOP) Function

This functions accepts an OOP for a Semaphore object and puts the current process to sleep, unless the semaphore has excess signals on it. Since a Smalltalk call-in is not

an atomic operation, the correct way to signal a semaphore is not to send the `wait` method to the object but, rather, to use:

```
asyncWait(semaphoreOOP)
```

The `sync` in the name of this function distinguishes it from `asyncSignal`, in that it cannot be called from within a signal handler.

`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);
incAddOOP(myNewObject);
obj = (myNewObjectData) oopToObj(myNewObject);
obj->arguments = objectAlloc(classNameToOOP("Array"), 10);
incAddOOP(obj->arguments);
...
```

The macros are:

mst_Object *oopToObj* (*OOP*) Macro

Dereference a pointer to an OOP into a pointer to the actual object data (see ⟨undefined⟩ [Object representation], page ⟨undefined⟩). The result of `oopToObj` 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 ⟨undefined⟩ [Smalltalk types], page ⟨undefined⟩).

mst_Boolean *oopIsReadOnly* (*OOP*) Macro

Answer whether or not the given OOP is read-only (see ⟨undefined⟩ [Special objects], page ⟨undefined⟩). Note that being read-only only limits access to indexed instance variables from Smalltalk code. Fixed instance variables, usage of `instVarAt:put:` and C accesses cannot be write-protected.

makeOOPReadOnly (*OOP*, *mst_Boolean*) Macro

Set whether or not the given OOP is read-only.

markOOPToFinalize (*OOP*, *mst_Boolean*) Macro

Set whether or not the given OOP is a 'finalizable' oop. (see ⟨undefined⟩ [Special objects], page ⟨undefined⟩).

makeOOPWeak (*OOP, mst_Boolean*) Macro
Set whether or not the given OOP is a 'weak' oop (see ⟨undefined⟩ [Special objects], page ⟨undefined⟩).

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

mst_Boolean *isClass* (*OOP, OOP*) Macro
Return whether the class of the OOP passed as the first parameter is the OOP passed as the second parameter.

mst_Boolean *isInt* (*OOP*) Macro
Return a Boolean indicating whether or not the OOP is an Integer object; the value of Integer objects is encoded directly in the OOP, not separately in a `mst_Object` structure. It is not safe to use `oopToObj` and `oopClass` if `isInt` returns false.

mst_Boolean *isOOP* (*OOP*) Macro
Return a Boolean indicating whether or not the OOP is a 'real' object (and not an Integer). It is safe to use `oopToObj` and `oopClass` only if `isOOP` returns true.

mst_Boolean *isNil* (*OOP*) Macro
Return a Boolean indicating whether or not the OOP points to `nil`, the undefined object.

mst_Boolean *arrayOOPAt* (*mst_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 `mst_Object` is defined, which prevents `indexedOOP` from working.

mst_Boolean *stringOOPAt* (*mst_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 `mst_Object` is defined, which prevents `indexedByte` from working.

mst_Boolean *indexedWord* (*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 ⟨undefined⟩ [Object representation], page ⟨undefined⟩).

mst_Boolean *indexedByte* (*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 ⟨undefined⟩ [Object representation], page ⟨undefined⟩).

mst_Boolean *indexedOOP* (*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 ⟨undefined⟩ [Object representation], page ⟨undefined⟩).

4.7 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
instanceVariableNames: 'database request'
classVariableNames: ''
poolDictionaries: ''
category: 'SQL-C interface'

 SQLAction subclass: #SQLRequest
instanceVariableNames: 'returnedRows'
classVariableNames: ''
poolDictionaries: ''
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 ⟨undefined⟩ [Smalltalk types], page ⟨undefined⟩, point to the actual object only indirectly) and get a pointer to the actual data. You do that with the `oopToObj` macro (note the type casting):

```
SQLAction action = (SQLAction) oopToObj(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;
...
invokeSQLQuery(
    oopToCObject(action->database),
    oopToString(action->request);
    sqlQueryCompletedCallback, /* Callback function */
    oop); /* Passed to the callback */

...
/* Imagine that invokeSQLQuery runs asynchronously and calls this
   when the job is done. */
void
sqlQueryCompletedCallback(result, database, request, clientData)
QueryResult *result;
DB *database;
char *request;
OOP clientData;
{
    SQLUpdateQuery query;
    OOP rows;
    OOP cObject;

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

    if (isClass(oop, sqlActionClass))
        return;

    if (isClass(oop, sqlObjectCreationClass)) {
        SQLObjectCreation oc;
        oc = (SQLObjectCreation) oopToObj(clientData);
        cObject = cObjectToOOP(result->dbObject)
        oc->newDBObject = cObject;
    } else {
        /* SQLRequest or SQLUpdateQuery */
```

```
            cObject = cObjectToOOP(result->rows);
            query = (SQLUpdateQuery) oopToObj(clientData);
            query->returnedRows = cObject;
            if (isClass(oop, sqlUpdateQueryClass)) {
                query->numReturnedRows = intToOOP(result->count);    <<<
            }
        }
        unregisterOOP(cObject); /* no need to force it alive */
    }
```

Note that the result of `oopToObj` 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 ⟨undefined⟩ [Smalltalk types], page ⟨undefined⟩). That's why I passed the OOP to the callback, not the object pointer itself.

Note that in the line marked '`<<<`' I did not have to reload the `query` variable because I used `intToOOP`. If I used any other function, I would have had to create all the required objects (using, if needed, the incubator described in ⟨undefined⟩ [Incubator], page ⟨undefined⟩), dereference the pointer, and store the data in the object.

Also, you should remember to unregister every object created with the "C to Smalltalk" fnctions (again, with the exception of `intToOOP`).

If your class has indexed instance variables, you can use the `indexedWord`, `indexedOOP` and `indexedByte` macros declared in `gstpub.h`, which return an lvalue for the given indexed instance variable—for more information, see ⟨undefined⟩ [Other C functions], page ⟨undefined⟩.

4.8 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 are two snippets from '`main.c`' and '`libgst/cint.c`'.

```
/* From main.c */
int main(argc, argv)
int     argc;
char    **argv;
{
  smalltalkArgs(argc, argv);
  initSmalltalk();
  topLevelLoop();

  exit(0);
```

```
}

/* From cint.c */
void initCFuncs()
{
  /* Access to command line args */
  defineCFunc("getArgc", getArgc);
  defineCFunc("getArgv", getArgv);

  /* Test functions */
  defineCFunc("testCallin", testCallin);
  defineCFunc("testCString", testCString);
  defineCFunc("testCStringArray", testCStringArray);

  /* ... */

  /* Initialize any user C function definitions. initUserCFuncs,
     defined in cfuncs.c, is overridden by explicit definition
     before linking with the Smalltalk library. */
  initUserCFuncs();
}
```

Your initialization code will be almost the same as that in GNU Smalltalk's `main()`, with the exception of the call to `topLevelLoop`. All you'll have to do is to pass some arguments to the GNU Smalltalk library via `smalltalkArgs`, and then call `initSmalltalk`.

Note that `initSmalltalk` will likely take some time (from a second to 30-40 seconds), because it has to check if the image file must be be rebuilt and, if so, it reloads and recompiles the 34000 lines of Smalltalk code in a basic image. To avoid this check, pass a '`-I`' flag:

```
char myArgv[][] = { "-I", "myprog.im", nil };
int myArgc;
/* ... */
myArgc = sizeof(myArgv) / sizeof (char *) - 1;
smalltalkArgs(myArgc, myArgv);
```

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 **without** `defineCFunc`-ing your own C call-outs, since GNU Smalltalk recalculates the addresses of the C call-outs every time it is started.

4.9 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 ⟨undefined⟩ [Smalltalk types], page ⟨undefined⟩ 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 *incAddOOP* (*OOP anOOP*) Macro
Adds a new object to the protected set.

IncPtr *incSavePointer* () 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 *incRestorePointer* (*IncPtr 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 incSavePointer using the incRestorePointer function.

Here's an example from cint.c:

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

```
desc = (CFuncDescriptor)
   newInstanceWith(cFuncDescriptorClass, numArgs);
desc->cFunction = cObjectNew(funcAddr);    // 1
desc->cFunctionName = stringNew(funcName); // 2
desc->numFixedArgs = fromInt(numArgs);
desc->returnType = classifyTypeSymbol(returnTypeOOP, true);
for (i = 1; i <= numArgs; i++) {
  desc->argTypes[i - 1] =
   classifyTypeSymbol(arrayAt(argsOOP, i), false);
}
```

```
return (allocOOP(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 = incSavePointer();
desc = (CFuncDescriptor)
  newInstanceWith(cFuncDescriptorClass, numArgs);
descOOP = allocOOP(desc);
incAddOOP(descOOP);

desc->cFunction = cObjectNew(funcAddr);
incAddOOP(desc->cFunction);

desc->cFunctionName = stringNew(funcName);
/* 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 = fromInt(numArgs);
desc->returnType = classifyTypeSymbol(returnTypeOOP, true);
for (i = 1; i <= numArgs; i++) {
  desc->argTypes[i - 1] =
   classifyTypeSymbol(arrayAt(argsOOP, i), false);
}

incRestorePointer(ptr);
return (descOOP);
```

Note that it is permissible for a couple of 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 incRestorePointer, 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.

5 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.

5.1 Getting started

5.1.1 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:

```
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.

5.1.2 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.[1]

[1] It also prints out a lot of statistics. Ignore these; they provide information on the performance of the underlying Smalltalk engine. You can inhibit them by starting Smalltalk as either:

```
$ gst -q
```

or

```
$ gst -r
```

5.1.3 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. 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.

5.1.4 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              !
```

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:

---

[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.

```
(9 + 7) printNl !
```

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

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

5.2 Using some of the Smalltalk classes

This chapter has examples which need a place to hold the objects they create. The following line creates such a place; for now, treat it as magic. At the end of the chapter we will revisit it with an explanation. Type in:

```
Smalltalk at: #x put: 0 !
```

Now let's create some new objects.

5.2.1 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:

[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.

```
(x at: 1) printNl !
```

which prints:

```
nil
```

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) printNl !
```

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) printNl !
```

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

5.2.2 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:

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

```
x printNl !
```

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 ()
```

Now let's toss some stuff into it. We'll add the numbers 5 and 7, plus the string 'foo'. We could type:

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

But let's save a little typing by using a Smalltalk shorthand:

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

This line does exactly what the previous example's three lines 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 printNl !
```

we'll see that it now contains our data:

```
Set (5 'foo' 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 !
x printNl !
```

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

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

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) printNl !
(x includes: 5) printNl !
```

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:

5.2.3 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:

```
x := Dictionary new.
x at: 'One' put: 1 !
x at: 'Two' put: 2 !
x at: 1 put: 'One' !
x 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:

```
(x at: 1) printNl !
(x at: 'Two') printNl !
```

which prints respectively:

```
'One'
2
```

We can also ask a dictionary to print itself:

```
x printNl !
```

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.

5.2.4 Smalltalk dictionary

If you'll remember from the beginning of the chapter, we started out by saying:

```
Smalltalk at: #x put: 0 !
```

This code should look familiar—the `at:put:` message is how we've been storing information in our own arrays and dictionaries. In a Smalltalk environment the name `Smalltalk` has been preset to point to a dictionary[6] which both you and Smalltalk can use. To see how this sharing works, we'll first try to use a variable which Smalltalk doesn't know about:

```
y := 0 !
```

Smalltalk complains because `y` is an unknown variable. Using our knowledge of dictionaries, and taking advantage of our access to Smalltalk's dictionary, we can add it ourselves:

[6] Actually, a SystemDictionary, which is just a Dictionary with some extra methods to run things when Smalltalk first starts and to do nice things with a Smalltalk environment

```
Smalltalk at: #y put: 0 !
```

The only mystery left is why we're using `#y` instead of our usual quoted string. This is one of those simple questions whose answer runs surprisingly deep. The quick answer is that `#y` and `'y'` are pretty much the same, except that the former will always be the same object each time you use it, whereas the latter can be a new string each time you do so.[7]

Now that we've added `y` to Smalltalk's dictionary, we try again:

```
y := 1 !
```

It works! Because you've added an entry for `y`, Smalltalk is now perfectly happy to let you use this new variable. If you have some spare time, you can print out the entire Smalltalk dictionary with:

```
Smalltalk printNl !
```

As you might suspect, this will print out quite a large list of names! If you get tired of watching Smalltalk grind it out, use your interrupt key (control-C, usually) to bring Smalltalk back to interactive mode.

5.2.5 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.

5.3 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.

5.3.1 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 ities; for instance, types of objects which may be compared as greater or less than each other fall under a class known as *Magnitude*.

[7] For more detail, See ⟨undefined⟩ [Two flavors of equality], page ⟨undefined⟩

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.

5.3.2 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.

5.3.3 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.

5.4 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:

```
Smalltalk 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 less than 700K bytes, and contains all variables, classes, and definitions you have added.

5.4.1 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
    instanceVariableNames: 'balance'
    classVariableNames: ''
    poolDictionaries: ''
    category: nil !
```

Quite a mouthful, isn't it? Most people end up customizing their editor to pop this up at a push of a button. But 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*. The other parts of the message can be ignored, but `instanceVariableNames: 'balance'` tells it that each object in this subclass will have a hidden variable named `balance`.[8]

5.4.2 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) printNl !
```

And your string is printed back to you. Try this with class Integer, too:

```
(Integer comment) printNl !
```

5.4.3 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:

[8] In case you're having a hard time making out the font, the '' after `classVariableNames:` and `poolDictionaries:` are a pair of single quotes—an empty string.

```
!Account class methodsFor: 'instance creation'!

new
    | r |

    r := super new.
    r init.
    ^r
! !
```

Again, programming your editor to do this is recommended. The important points about this are:

- `Account class` means that we are defining messages which are to be sent to the Account class itself.
- `methodsFor: 'instance creation'` is more documentation support; it says that all of the methods defined will be to support 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!

This is also the first example where we've had to use more than one statement, and thus a good place to present the statement separator—the `.` period. 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.

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.

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.[9]

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.

5.4.4 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 methodsFor: 'instance initialization'!
init
    balance := 0
! !
```

It looks quite a bit like the previous method definition, except that the first one said `Account class methodsFor:...`, and ours says `Account methodsFor:...`.

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.[10]

5.4.5 Looking at our Account

Let's create an instance of class Account:

```
Smalltalk at: #a put: (Account new) !
```

[9] 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.

[10] 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.

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. Let's take a look at the Account object we just created:

```
a printNl !
```

It prints:

```
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 methodsFor: 'printing'!
printOn: stream
    super printOn: stream.
    stream nextPutAll: ' with balance: '.
    balance printOn: stream
! !
```

Now give it a try again:

```
a printNl !
```

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.

5.4.6 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 methodsFor: 'moving money'!
spend: amount
    balance := balance - amount
!
deposit: amount
    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 printNl!
a spend: 10!
a printNl!
```

5.4.7 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.

5.5 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.

5.5.1 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
    instanceVariableNames: 'interest'
    classVariableNames: ''
    poolDictionaries: ''
    category: nil !
```

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.

```
!Savings methodsFor: 'initialization'!
init
    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:

```
!Savings methodsFor: 'interest'!
interest: amount
    interest := interest + amount.
    self deposit: amount
!
clearInterest
    | oldinterest |

    oldinterest := interest.
    interest := 0.
    ^oldinterest
! !
```

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.[11]

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.[12]

5.5.2 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

We will define this as another subclass of Account:

---

[11] `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.

[12] 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.

```
Account subclass: #Checking
     instanceVariableNames: 'checknum checksleft'
     classVariableNames: ''
     poolDictionaries: ''
     category: nil !
```

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.

```
!Checking methodsFor: 'Initialization'!
init
     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.

5.5.3 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:

```
!Checking methodsFor: 'spending'!
newChecks: number count: checkcount
     checknum := number.
     checksleft := checkcount
!

writeCheck: amount
     | 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:

```
Smalltalk at: #c put: (Checking new) !
c printNl !
```

```
c deposit: 250 !
c printNl !
c newChecks: 100 count: 50 !
c printNl !
(c writeCheck: 32) printNl !
c printNl !
```

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.

5.6 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.

5.6.1 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 methodsFor: 'spending'!
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.

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.[13]

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 ] !
```

5.6.2 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.

[13] 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.

```
Account subclass: #Checking
    instanceVariableNames: 'checknum checksleft history'
    classVariableNames: ''
    poolDictionaries: ''
    category: nil !
```

This is the same syntax as the last time we defined a checking account, except that we have three instance variables: the `checknum` and `checksleft` which have always been there, and our new `history` variable; since we have removed no instance variables, the old method 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:

```
!Checking methodsFor: 'initialization'!
init
    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.

```
!Checking methodsFor: 'spending'!
writeCheck: amount
    | 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.[14] 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.

```
!Checking methodsFor: 'printing'!
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

[14] 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 over-kill) to store a dictionary as the value—and then store as many pieces of information as you'd like under each slot!

to come back and re-write this code so long as the object in the dictionary honors our `printNl`/`printOn:` messages sages.

```
!Checking methodsFor: 'printing'!
printChecks
    history associationsDo: [ :assoc |
        (assoc key) print.
        ' - ' print.
        (assoc value) printNl.
    ]
! !
```

We still see a code block object being passed to the dictionary, but `:assoc |` is something new. A code block can optionally receive arguments. In this case, the argument is the key/value pair, known in Smalltalk as an *Association*. This is the way that a dictionary object stores its key/value pairs internally. In fact, when you sent an at:put: message to a dictionary object, the first thing it does is pack them into a new object from the Association class. 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.

Our code merely uses the `key` and `value` messages to ask the association for the two values. 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 on the relationship between an Association and the argument to a code block. In this example, we passed a `associationsDo:` message to a dictionary. A dictionary invokes the passed code block with an Association when processing an associationsDo: message. But code blocks can receive any type of argument: the type is determined by the code which invokes the code block; Dictionary's `associationDo:` method, in this case. 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.

5.7 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.[15] 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.

5.7.1 Integer loops

Integer loops are constructed by telling a number to drive the loop. Try this example to count from 1 to 20:

[15] 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 `associationsDo:` message works only with dictionaries. The difference is that `do:` passes only the value portion, while `associationsDo:` passes the entire key/value pair in an Association object.

```
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 ] !
```

5.7.2 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.

```
Smalltalk at: #i put: (Interval from: 5 to: 10) !
i printNl !
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 printNl !
i do: [:x| x printNl] !
```

5.7.3 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 methodsFor: 'scanning'!
checksOver: amount do: aBlock
    history associationsDo: [:assoc|
        ((assoc value) > amount)
               ifTrue: [aBlock value: (assoc 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: (assoc key)]
```

invokes the user-supplied block, passing as an argument the association's 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.[16]

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

[16] There is also a `valueWithArguments:` message which accepts an array holding as many arguments as you would like.

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:

```
Smalltalk at: #mycheck put: (Checking new) !
mycheck deposit: 250 !
mycheck newChecks: 100 count: 40 !
mycheck writeCheck: 10 !
mycheck writeCheck: 52 !
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 methodsFor: 'scanning'!
checksOver: amount do: aBlock
    | chosen |
    chosen := history select: [:amt| amt > amount].
    chosen associationsDo: aBlock
! !
```

Unlike our previous definition of `checksOver:do:`, this one passes the user's code block the association, not just a check number. How could this code be rewritten to remedy this, while still using select:?

Yet, this new behavior can be useful. You can use the same set of tests that we ran above. Notice that our code block:

```
[:x| x printNl]
```

now prints out an Association. This has a very nice effect: with our old method, we were told which check numbers were above a given amount; with this new method, we get the check number and amount in the form of an Association. When we print an association, since the key is the check number and the value is the check amount, we get a list of checks over the amount in the format:

```
CheckNum -> CheckVal
```

5.8 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.

5.8.1 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.

5.8.2 Nested Calls

Type the following lines:

```
Smalltalk at: #x put: (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...

5.8.3 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.[17]

Thus:

```
Smalltalk at: #x put: (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.

[17] When using the Blox GUI, it actually pops up a so-called *Inspector window*.

5.9 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.

5.9.1 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.[18].

```
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
```

[18] This listing is courtesy of the printHierarchy method supplied by GNU Smalltalk author Steve Byrne. It's in the '`kernel/Browser.st`' file.

```
            CByte
               CBoolean
         CUInt
         CULong
         CUShort
Collection
   Bag
   MappedCollection
   SequenceableCollection
      ArrayedCollection
         Array
         ByteArray
         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 ⟨undefined⟩ [The truth on metaclasses], page ⟨undefined⟩.

5.9.2 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?

```
Array variableSubclass: #NiledArray
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
```

```
    category: nil !

!NiledArray methodsFor: 'bounds checking'!
boundsCheck: index
    ^(index < 1) | (index > (self basicSize))
! !

!NiledArray methodsFor: 'basic'!
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. Instead of our usual `subclass:` message, we use a `variableSubclass:` message. This reflects the 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?[19]

`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

[19] 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.

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 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.

5.9.3 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
    instanceVariableNames: 'realpart imagpart'
    classVariableNames: ''
    poolDictionaries: ''
    category: nil !
!Complex class methodsFor: 'creating'!
new
    ^self error: 'use real:imaginary:'
!
new: ignore
    ^self new
!
real: r imaginary: i
    ^(super new) setReal: r setImag: i
```

```
! !

!Complex methodsFor: 'creating--private'!
setReal: r setImag: i
    realpart := r.
    imagpart := i.
    ^self
! !

!Complex methodsFor: 'basic'!
real
    ^realpart
!
imaginary
    ^imagpart
! !

!Complex methodsFor: 'math'!
+ val
    ^Complex real: (realpart + val real)
        imaginary: (imagpart + val imaginary)
!
- val
    ^Complex real: (realpart - val real)
        imaginary: (imagpart - val imaginary)
!
* val
    ^Complex real: (realpart * val real) - (imagpart * val imaginary)
        imaginary: (imagpart * val real) + (realpart * val imaginary)
!
/ val
    | 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
! !

!Complex methodsFor: 'comparison'!

= val
    ^(realpart = val real) & (imagpart = val imaginary)
!
> val
    ^self shouldNotImplement
!
>= val
    ^self shouldNotImplement
!
```

```
< val
    ^self shouldNotImplement
!
<= val
    ^self shouldNotImplement
! !

!Complex methodsFor: 'printing'!
printOn: aStream
    aStream nextPut: $(.
    realpart printOn: aStream.
    aStream nextPut: $,.
    imagpart printOn: aStream.
    aStream nextPut: $)
! !
```

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, 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?

5.9.4 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:

```
Smalltalk at: #a put: (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 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.

5.10 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.

5.10.1 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`[20].

5.10.2 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:

```
Smalltalk at: #a put: (Array new: 10) !
a at: 4 put: 1234 !
a at: 9 put: 5678 !
Smalltalk at: #s put: (ReadWriteStream on: a) !
s inspect !
s position: 1 !
s inspect !
```

[20] Try executing it under Blox, where the Transcript is linked to the omonymous window!

```
s nextPut: 11; nextPut: 22 !
(a at: 1) printNl !
a do: [:x| x printNl] !
s position: 2 !
s do: [:x| x printNl] !
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:

```
Smalltalk at: #s put: (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) printNl !
```

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:

```
Smalltalk at: #s put: (ReadWriteStream on: (Array new: 0)) !
s next: 4 put: 'Hi!' !
s position: 1 !
s do: [:x | x printNl] !
```

5.10.3 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:

```
Smalltalk at: #f put: (FileStream
    open: '/etc/passwd'
    mode: FileStream read) !
f do: [ :c | Transcript nextPut: c ] !
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' !
```

5.10.4 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.

5.11 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...

5.11.1 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.[21] 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 creating a class.

For most example classes, we've used the message:

```
subclass:
instanceVariableNames:
classVariableNames:
poolDictionaries:
category:
```

But when we implemented our CheckedArray example, we used `variableSubclass:` instead of just `subclass:`. The choice of these two kinds of class creation (and two more we'll show shortly) defines the fundamental structure of Smalltalk objects created within a given class. Let's consider the differences in the next sub-sections.

subclass:. This kind of class creation 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.[22].

variableSubclass:. All the other types of class are a superset of a `subclass:`. 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.

[21] For GNU Smalltalk, the size of a C `long`, which is usually 32 bits.

[22] C requires one or more; zero is allowed in Smalltalk

variableByteSubclass:. This is a special case of `variableSubclass:`; the storage age allocated as specified by new: is an array of bytes. The analog in C would be a dynamically allocated structure with scalar fields[23], followed by a array of `char`.

variableWordSubclass:. Once again, this is a special case of `variableSubclass:`; the storage age allocated as specified by new: is an array of C signed 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.

Accessing These 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 the at: and at:put: messages. Consider a Dictionary—it is a `variableSubclass:` type of 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.

An Example. 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 variableSubclass: 'RangedArray'
    instanceVariableNames: 'base'
    classVariableNames: ''
    poolDictionaries: ''
    category: nil !
RangedArray comment: 'I am an Array whose base is arbitrary' !
!RangedArray class methodsFor: 'creation'!
new
    ^self error: 'Use new:base:'
!
new: size
    ^self new: size base: 1
!
new: size base: b
    ^(super new: size) init: b
! !
!RangedArray methodsFor: 'init'!
init: b
    base := (b - 1).   "- 1 because basicAt: works with a 1 base"
    ^self
! !
!RangedArray methodsFor: 'basic'!
rangeCheck: i
```

[23] This is not always true for other Smalltalk implementations, who don't allow instance variables in variableByteSubclasses and variableWordSubclasses.

```
    ((i <= base) | (i > (base + (self basicSize)))) ifTrue: [
        'Bad index value: ' printOn: stderr.
        i printOn: stderr.
        (Character nl) printOn: stderr.
        ^self error: 'illegal index'
    ]
!
at: i
    self rangeCheck: i.
    ^self basicAt: (i-base)
!
at: i put: v
    self rangeCheck: i.
    ^self basicAt: (i-base) 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:

```
Smalltalk at: #a put: (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 methodsFor: 'basic'!
do: aBlock
    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[24], 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

[24] Actually, in GNU Smalltalk `do:` is not the only message assuming that.

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.[25]

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.

Basic Allocation. In this chapter, we've seen the fundamental mechanisms used to allocate and index storage. When the storage need not be accessed with peak 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. When you move into this area of object development, inheritance and polymorphism become trickier; each level must coordinate its use of the underlying array with other levels.

5.11.2 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

[25] Some of these classes actually redefine `do:` for performance reasons, but they would work even if the parent class' implementation of `do:` was kept.

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 =:

```
Smalltalk at: #x put: 0 !
Smalltalk at: #y put: 0 !
x := 'Hello' !
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.

5.11.3 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; type 'gst -r' so that you have no run-time statistics on the screen; type the text printed in `mono-spaced` font.

```
st> ^Set superclass!
returned value is Collection

st> ^Collection superclass!
returned value is Object

st> ^Object superclass!
returned value is nil
```

Nothing new for now. Let's try something else:

```
st> ^#(1 2 3) class!
returned value is Array

st> ^'123' class!
returned value is String

st> ^Set class!
```

returned value is Set class

```
st> ^Set class class!
returned value is Metaclass
```

You get it, that strange `Set class` thing is something called "a meta-class"... let's go on:

```
st> ^Set class superclass!
returned value is Collection class

st> ^Collection class superclass!
returned value is 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!
returned value is 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!
returned value is ClassDescription

st> ^ClassDescription superclass!
returned value is Behavior

st> ^Behavior superclass!
returned value is Object
```

Class is a subclass of other classes. `ClassDescription` is abstract; `Behavior` is concrete but has 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 |
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
returned value is nil
```

If you send `#class` repeatedly, it seems that you end up in a loop made of class `Metaclass`[26] 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.

5.11.4 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[27], once observed that implementing

[26] Which turns out to be another subclass of `ClassDescription`.

[27] And like the one that GNU Smalltalk includes as an experimental feature.

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 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 number 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:`[28]

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 GST 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`'.

5.12 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 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 and TCP/IP.

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!

5.13 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 [prim] [temps] exprs
message: id | binsel id | [keysel id]+
prim: ‘‘<’’ ‘‘primitive:’’ number ‘‘>’’
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 primitive; it has not been covered in this tutorial; it provides an 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: selchar[selchar]
```

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
arrayconst: ‘‘#’’ array | ‘‘#’’ bytearray
bytearray: ‘‘[’’ [number]* ‘‘]’’
array: ‘‘(’’ [literal | array | bytearray | ]* ‘‘)’’
number: [[dig]+ ‘‘r’’] [‘‘-’’] [alphanum]+ [‘‘.’’ [alphanum]+]
[‘‘e’’[‘‘-’’][dig]+].
string: "’"[char]*"’"
charconst: ‘‘$’’char
symconst: ‘‘#’’symbol
```

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:

```
Smalltalk at: #a put: #(1 2 'Hi' $x $Hello 4 26r5H) !
```

There are also ByteArray constants, whose elements are constrained to be integers between 0 and 255; they would look like:

```
Smalltalk at: #a put: #[1 2 34 16r8F 26r3H 253] !
```

`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:
```

```
id: letter[letter|dig]*
selchar: ‘‘+’’ | ‘‘-’’ | ‘‘*’’ | ‘‘/’’ | ‘‘~’’ | ‘‘|’’ | ‘‘,’’ |
‘‘<’’ | ‘‘>’’ | ‘‘=’’ | ‘‘&’’ | ‘‘ʃ | ‘‘?’’
alphanum: ‘‘0’’..‘‘9’’ | ‘‘A’’..‘‘Z’’
dig: ‘‘0’’..‘‘9’’
```

These are the categories of characters and how they are combined at the most basic level. selchar simply lists the characters which can be combined to name a binary message (binary messages including a question mark are almost never used).

6 Class reference

6.1 AlternativeObjectProxy

Defined in namespace Smalltalk
Category: Streams-Files

I am a proxy that uses the same ObjectDumper to store an object which is not the object to be dumped, but from which the dumped object can be reconstructed. I am an abstract class, using me would result in infinite loops because by default I try to store the same object again and again. See the method comments for more information

6.1.1 AlternativeObjectProxy class: instance creation

acceptUsageForClass: aClass

The receiver was asked to be used as a proxy for the class aClass. Answer whether the registration is fine. By default, answer true except if AlternativeObjectProxy itself is being used.

on: anObject

Answer a proxy to be used to save anObject. IMPORTANT: this method MUST be overridden so that the overridden version sends #on: to super passing an object that is NOT the same as anObject (alternatively, you can override #dumpTo:, which is what NullProxy does), because that would result in an infinite loop! This also means that AlternativeObjectProxy must never be used directly – only as a superclass.

6.1.2 AlternativeObjectProxy: accessing

object

Reconstruct the object stored in the proxy and answer it. A subclass will usually override this

object: theObject

Set the object to be dumped to theObject. This should not be overridden.

primObject

Reconstruct the object stored in the proxy and answer it. This method must not be overridden

6.2 ArithmeticError

Defined in namespace Smalltalk
Category: Language-Exceptions

An ArithmeticError exception is raised by numeric classes when a program tries to do something wrong, such as extracting the square root of a negative number.

6.2.1 ArithmeticError: description

description
Answer a textual description of the exception.

6.3 Array

Defined in namespace Smalltalk
Category: Collections-Sequenceable
My instances are objects that have array-like properties: they are directly indexable by integers starting at 1, and they are fixed in size. I inherit object creation behavior messages such as #with:, as well as iteration and general access behavior from SequenceableCollection.

6.3.1 Array: mutating objects

multiBecome: anArray
Transform every object in the receiver in each corresponding object in anArray. anArray and the receiver must have the same size

6.3.2 Array: printing

printOn: aStream
Print a representation for the receiver on aStream

6.3.3 Array: testing

isArray Answer 'true'.

6.4 ArrayedCollection

Defined in namespace Smalltalk
Category: Collections-Sequenceable
My instances are objects that are generally fixed size, and are accessed by an integer index. The ordering of my instance's elements is determined externally; I will not rearrange the order of the elements.

6.4.1 ArrayedCollection class: instance creation

new Answer an empty collection

new: size withAll: anObject
Answer a collection with the given size, whose elements are all set to anObject

with: element1
Answer a collection whose only element is element1

with: element1 with: element2
Answer a collection whose only elements are the parameters in the order they were passed

with: element1 with: element2 with: element3
Answer a collection whose only elements are the parameters in the order they were passed

with: element1 with: element2 with: element3 with: element4
Answer a collection whose only elements are the parameters in the order they were passed

with: element1 with: element2 with: element3 with: element4 with: element5
Answer a collection whose only elements are the parameters in the order they were passed

withAll: aCollection
Answer a collection whose elements are the same as those in aCollection

6.4.2 ArrayedCollection: basic

, aSequenceableCollection
Answer a new instance of an ArrayedCollection containing all the elements in the receiver, followed by all the elements in aSequenceableCollection

add: value This method should not be called for instances of this class.

copyFrom: start to: stop
Answer a new collection containing all the items in the receiver from the start-th and to the stop-th

copyWith: anElement
Answer a new instance of an ArrayedCollection containing all the elements in the receiver, followed by the single item anElement

copyWithout: oldElement
Answer a copy of the receiver to which all occurrences of oldElement are removed

6.4.3 ArrayedCollection: built ins

size Answer the size of the receiver

6.4.4 ArrayedCollection: copying Collections

reverse Answer the receivers' contents in reverse order

6.4.5 ArrayedCollection: enumerating the elements of a collection

collect: aBlock
Answer a new instance of an ArrayedCollection containing all the results of evaluating aBlock passing each of the receiver's elements

reject: aBlock
Answer a new instance of an ArrayedCollection containing all the elements in the receiver which, when passed to aBlock, answer false

select: aBlock
Answer a new instance of an ArrayedCollection containing all the elements in the receiver which, when passed to aBlock, answer true

with: aSequenceableCollection collect: aBlock
Evaluate aBlock for each pair of elements took respectively from the re- ceiver and from aSequenceableCollection; answer a collection of the same kind of the receiver, made with the block's return values. Fail if the receiver has not the same size as aSequenceableCollection.

6.4.6 ArrayedCollection: storing

storeOn: aStream
Store Smalltalk code compiling to the receiver on aStream

6.5 Association

Defined in namespace Smalltalk
Category: Language-Data types
My instances represent a mapping between two objects. Typically, my "key" object is a symbol, but I don't require this. My "value" object has no conventions associated with it; it can be any object at all.

6.5.1 Association class: basic

key: aKey value: aValue
Answer a new association with the given key and value

6.5.2 Association: accessing

key: aKey value: aValue
Set the association's key to aKey, and its value to aValue

value Answer the association's value

value: aValue
Set the association's value to aValue

6.5.3 Association: printing

printOn: aStream
Put on aStream a representation of the receiver

6.5.4 Association: storing

storeOn: aStream
Put on aStream some Smalltalk code compiling to the receiver

6.5.5 Association: testing

= anAssociation
Answer whether the association's key and value are the same as anAssociation's, or false if anAssociation is not an Association

hash Answer an hash value for the receiver

6.6 Autoload

Defined in namespace Smalltalk
Category: Examples-Useful tools
I am not a part of the normal Smalltalk kernel class system. I provide the ability to do late-loading or "on demand loading" of class definitions. Through me, you can define any class to be loaded when any message is sent to the class itself (such as to create an instance).

6.6.1 Autoload class: instance creation

class: classNameString from: fileNameString
Make Smalltalk automatically load the class named classNameString from fileNameString when needed

6.6.2 Autoload: accessing

doesNotUnderstand: aMessage
Load the file, then reinvoke the method forwarding it to the newly loaded class.

6.7 Bag

Defined in namespace Smalltalk
Category: Collections-Unordered
My instances are unordered collections of objects. You can think of me as a set with a memory; that is, if the same object is added to me twice, then I will report that that element has been stored twice.

6.7.1 Bag class: basic

new Answer a new instance of the receiver

new: size Answer a new instance of the receiver, with space for size distinct objects

6.7.2 Bag: Adding to a collection

add: newObject
Add an occurrence of newObject to the receiver. Answer newObject

add: newObject withOccurrences: anInteger
If anInteger > 0, add anInteger occurrences of newObject to the receiver. If anInteger < 0, remove them. Answer newObject

6.7.3 Bag: enumerating the elements of a collection

asSet Answer a set with the elements of the receiver

do: aBlock
Evaluate the block for all members in the collection.

6.7.4 Bag: extracting items

sortedByCount
Answer a collection of counts with elements, sorted by decreasing count.

6.7.5 Bag: printing

printOn: aStream
Put on aStream a representation of the receiver

6.7.6 Bag: Removing from a collection

remove: oldObject ifAbsent: anExceptionBlock
Remove oldObject from the collection and return it. If can't be found, answer instead the result of evaluationg anExceptionBlock

6.7.7 Bag: storing

storeOn: aStream
Put on aStream some Smalltalk code compiling to the receiver

6.7.8 Bag: testing collections

= aBag Answer whether the receiver and aBag contain the same objects

hash Answer an hash value for the receiver

includes: anObject
Answer whether we include anObject

occurrencesOf: anObject
Answer the number of occurrences of anObject found in the receiver

size Answer the total number of objects found in the receiver

6.8 Behavior

Defined in namespace Smalltalk
Category: Language-Implementation
I am the parent class of all "class" type methods. My instances know about the subclass/superclass relationships between classes, contain the description that instances are created from, and hold the method dictionary that's associated with each class. I provide methods for compiling methods, modifying the class inheritance hierarchy, examining the method dictionary, and iterating over the class hierarchy.

6.8.1 Behavior class: C interface

defineCFunc: cFuncNameString
withSelectorArgs: selectorAndArgs forClass: aClass returning: returnTypeSymbol args: argsArray Lookup the part on the C interface in this manual – it is too complex to describe it here ;-) Anyway this is private and kept for backward com- patibility. You should use defineCFunc:withSelectorArgs:returning:args:.

6.8.2 Behavior: accessing class hierarchy

allSubclasses
Answer the direct and indirect subclasses of the receiver in a Set

allSuperclasses
Answer all the receiver's superclasses in a collection

subclasses Answer the direct subclasses of the receiver in a Set

superclass Answer the receiver's superclass (if any, otherwise answer nil)

withAllSubclasses
Answer a Set containing the receiver together with its direct and indirect subclasses

withAllSuperclasses
Answer the receiver and all of its superclasses in a collection

6.8.3 Behavior: accessing instances and variables

allClassVarNames
: Return all the class variables understood by the receiver

allInstances
: Returns a set of all instances of the receiver

allInstVarNames
: Answer the names of every instance variables the receiver contained in the receiver's instances

allSharedPools
: Return the names of the shared pools defined by the class and any of its superclasses

classPool
: Answer the class pool dictionary. Since Behavior does not support classes with class variables, we answer an empty one; adding variables to it results in an error.

classVarNames
: Answer all the class variables for instances of the receiver

instanceCount
: Return a count of all the instances of the receiver

instVarNames
: Answer an Array containing the instance variables defined by the receiver

sharedPools
: Return the names of the shared pools defined by the class

subclassInstVarNames
: Answer the names of the instance variables the receiver inherited from its superclass

6.8.4 Behavior: accessing the methodDictionary

>> selector
: Return the compiled method associated with selector, from the local method dictionary. Error if not found.

allSelectors
: Answer a Set of all the selectors understood by the receiver

compiledMethodAt: selector
: Return the compiled method associated with selector, from the local method dictionary. Error if not found.

selectorAt: method
: Return selector for the given compiledMethod

selectors
: Answer a Set of the receiver's selectors

sourceCodeAt: selector
Answer source code (if available) for the given compiledMethod

sourceMethodAt: selector
This is too dependent on the original implementation

6.8.5 Behavior: browsing

getAllMethods
Answer the receiver's complete method dictionary - including inherited and not overridden methods. Each value in the dictionary is an Association, whose key is the class which defines the method, and whose value is the actual CompiledMethod

getDirectMethods
Answer the receiver's method dictionary; each value in the dictionary is not a CompiledMethod, but an Association, whose key is the class which defines the method (always the receiver), and whose value is the actual CompiledMethod

getIndirectMethods
Answer a dictionary of the receiver's inherited and not overridden methods. Each value in the dictionary is an Association, whose key is the class which defines the method, and whose value is the actual CompiledMethod

getMethods
Answer the receiver's complete method dictionary - including inherited and not overridden methods

getMethodsFor: aSelector
Get a dictionary with all the definitions of the given selector along the hierarchy. Each key in the dictionary is a class which defines the method, and each value in the dictionary is an Association, whose key is the class again, and whose value is the actual CompiledMethod

methodDictionary
Answer the receiver's method dictionary

newGetMethods
Answer the receiver's complete method dictionary - including inherited and not overridden methods. Each value in the dictionary is an Association, whose key is the class which defines the method, and whose value is the actual CompiledMethod

6.8.6 Behavior: built ins

basicNew Create a new instance of a class with no indexed instance variables; this method must not be overridden.

basicNew: numInstanceVariables
Create a new instance of a class with indexed instance variables. The instance has numInstanceVariables indexed instance variables; this method must not be overridden.

basicNewInFixedSpace
: Create a new instance of a class with no indexed instance variables. The instance is guaranteed not to move across garbage collections. Like #basicNew, this method should not be overridden.

basicNewInFixedSpace: numInstanceVariables
: Create a new instance of a class with indexed instance variables. The instance has numInstanceVariables indexed instance variables. The instance is guaranteed not to move across garbage collections. Like #basicNew:, this method should not be overridden.

compileString: aString
: Compile the code in aString, with no category. Fail if the code does not obey Smalltalk syntax. Answer the generated CompiledMethod if it does

compileString: aString ifError: aBlock
: Compile the code in aString, with no category. Evaluate aBlock (passing the file name, line number and description of the error) if the code does not obey Smalltalk syntax. Answer the generated CompiledMethod if it does

flushCache
: Invalidate the method cache kept by the virtual machine. This message should not need to be called by user programs.

makeDescriptorFor: funcNameString
: returning: returnTypeSymbol withArgs: argsArray Private - Answer a CFunctionDescriptor

methodsFor: category ifTrue: condition
: Compile the following code inside the receiver, with the given category, if condition is true; else ignore it

new Create a new instance of a class with no indexed instance variables

new: numInstanceVariables
: Create a new instance of a class with indexed instance variables. The instance has numInstanceVariables indexed instance variables.

someInstance
: Private - Answer the first instance of the receiver in the object table

6.8.7 Behavior: compilation (alternative)

methods Don't use this, it's only present to file in from Smalltalk/V

methodsFor
: Don't use this, it's only present to file in from Dolphin Smalltalk

methodsFor: category ifFeatures: features
: Start compiling methods in the receiver if this implementation of Smalltalk has the given features, else skip the section

methodsFor: category stamp: notUsed
: Don't use this, it's only present to file in from Squeak

privateMethods
Don't use this, it's only present to file in from IBM Smalltalk

publicMethods
Don't use this, it's only present to file in from IBM Smalltalk

6.8.8 Behavior: compiling methods

methodsFor: aCategoryString
Calling this method prepares the parser to receive methods to be compiled and installed in the receiver's method dictionary. The methods are put in the category identified by the parameter.

6.8.9 Behavior: creating a class hierarchy

addSubclass: aClass
Add aClass asone of the receiver's subclasses.

removeSubclass: aClass
Remove aClass from the list of the receiver's subclasses

superclass: aClass
Set the receiver's superclass.

6.8.10 Behavior: creating method dictionary

addSelector: selector withMethod: compiledMethod
Add the given compiledMethod to the method dictionary, giving it the passed selector. Answer compiledMethod

compile: code
Compile method source. If there are parsing errors, answer nil. Else, return a CompiledMethod result of compilation

compile: code ifError: block
Compile method source. If there are parsing errors, invoke exception block, 'block' passing file name, line number and error. description. Return a CompiledMethod result of compilation

compile: code notifying: requestor
Compile method source. If there are parsing errors, send #error: to the requestor object, else return a CompiledMethod result of compilation

compileAll
Recompile all selectors in the receiver. Ignore errors.

compileAll: aNotifier
Recompile all selectors in the receiver. Notify aNotifier by sen- ding #error: messages if something goes wrong.

compileAllSubclasses
Recompile all selector of all subclasses. Notify aNotifier by sen- ding #error: messages if something goes wrong.

compileAllSubclasses: aNotifier
Recompile all selector of all subclasses. Notify aNotifier by sen- ding #error: messages if something goes wrong.

createGetMethod: what
Create a method accessing the variable 'what'.

createGetMethod: what default: value
Create a method accessing the variable 'what', with a default value of 'value', using lazy initialization

createSetMethod: what
Create a method which sets the variable 'what'.

decompile: selector
Decompile the bytecodes for the given selector.

defineCFunc: cFuncNameString
withSelectorArgs: selectorAndArgs returning: returnTypeSymbol args: argsArray Lookup the C interface in the manual. Too complex to describe it here ;-)

edit: selector
Open Emacs to edit the method with the passed selector, then compile it

methodDictionary: aDictionary
Set the receiver's method dictionary to aDictionary

recompile: selector
Recompile the given selector, answer nil if something goes wrong or the new CompiledMethod if everything's ok.

recompile: selector notifying: aNotifier
Recompile the given selector. If there are parsing errors, send #error: to the aNotifier object, else return a CompiledMethod result of compilation

removeSelector: selector
Remove the given selector from the method dictionary, answer the Compiled-Method attached to that selector

removeSelector: selector ifAbsent: aBlock
Remove the given selector from the method dictionary, answer the Compiled-Method attached to that selector. If the selector cannot be found, answer the result of evaluating aBlock.

6.8.11 Behavior: enumerating

allInstancesDo: aBlock
Invokes aBlock for all instances of the receiver

allSubclassesDo: aBlock
Invokes aBlock for all subclasses, both direct and indirect.

allSubinstancesDo: aBlock
Invokes aBlock for all instances of each of the receiver's subclasses.

allSuperclassesDo: aBlock
Invokes aBlock for all superclasses, both direct and indirect.

selectSubclasses: aBlock
Return a Set of subclasses of the receiver satisfying aBlock.

selectSuperclasses: aBlock
Return a Set of superclasses of the receiver satisfying aBlock.

subclassesDo: aBlock
Invokes aBlock for all direct subclasses.

withAllSubclassesDo: aBlock
Invokes aBlock for the receiver and all subclasses, both direct and indirect.

withAllSuperclassesDo: aBlock
Invokes aBlock for the receiver and all superclasses, both direct and indirect.

6.8.12 Behavior: evaluating

evalString: aString to: anObject
Answer the stack top at the end of the evaluation of the code in aString. The code is executed as part of anObject

evalString: aString to: anObject ifError: aBlock
Answer the stack top at the end of the evaluation of the code in aString. If aString cannot be parsed, evaluate aBlock (see compileString:ifError:). The code is executed as part of anObject

evaluate: code
Evaluate Smalltalk expression in 'code' and return result.

evaluate: code ifError: block
Evaluate 'code'. If a parsing error is detected, invoke 'block'

evaluate: code notifying: requestor
Evaluate Smalltalk expression in 'code'. If a parsing error is encountered, send #error: to requestor

evaluate: code to: anObject
Evaluate Smalltalk expression as part of anObject's method definition

evaluate: code to: anObject ifError: block
Evaluate Smalltalk expression as part of anObject's method definition. This method is used to support Inspector expression evaluation. If a parsing error is encountered, invoke error block, 'block'

6.8.13 Behavior: hierarchy browsing

printHierarchy
Print my entire subclass hierarchy on the terminal.

printHierarchyEmacs
Print my entire subclass hierarchy on the terminal, in a format suitable for Emacs parsing.

6.8.14 Behavior: instance creation

newInFixedSpace
Create a new instance of a class without indexed instance variables. The instance is guaranteed not to move across garbage collections. If a subclass overrides #new, the changes will apply to this method too.

newInFixedSpace: numInstanceVariables
Create a new instance of a class with indexed instance variables. The instance has numInstanceVariables indexed instance variables. The instance is guaranteed not to move across garbage collections. If a subclass overrides #new:, the changes will apply to this method too.

6.8.15 Behavior: instance variables

addInstVarName: aString
Add the given instance variable to instance of the receiver

removeInstVarName: aString
Remove the given instance variable from the receiver and recompile all of the receiver's subclasses

6.8.16 Behavior: support for lightweight classes

article Answer an article ('a' or 'an') which is ok for the receiver's name

asClass Answer the first superclass that is a full-fledged Class object

environment
Answer the namespace that this class belongs to - the same as the superclass, since Behavior does not support namespaces yet.

name Answer the class name; this prints to the name of the superclass enclosed in braces. This class name is used, for example, to print the receiver.

nameIn: aNamespace
Answer the class name when the class is referenced from aNamespace - a dummy one, since Behavior does not support names.

6.8.17 Behavior: testing the class hierarchy

inheritsFrom: aClass
Returns true if aClass is a superclass of the receiver

kindOfSubclass
Return a string indicating the type of class the receiver is

6.8.18 Behavior: testing the form of the instances

instSize Answer how many fixed instance variables are reserved to each of the receiver's instances

isBits Answer whether the instance variables of the receiver's instances are bytes or words

isBytes Answer whether the instance variables of the receiver's instances are bytes

isFixed Answer whether the receiver's instances have no indexed instance variables

isIdentity Answer whether x = y implies x == y for instances of the receiver

isImmediate
Answer whether, if x is an instance of the receiver, x copy == x

isPointers Answer whether the instance variables of the receiver's instances are objects

isVariable Answer whether the receiver's instances have indexed instance variables

isWords Answer whether the instance variables of the receiver's instances are words

6.8.19 Behavior: testing the method dictionary

canUnderstand: selector
Returns true if the instances of the receiver understand the given selector

hasMethods
Return whether the receiver has any methods defined

includesSelector: selector
Returns true if the local method dictionary contains the given selector

scopeHas: name ifTrue: aBlock
If methods understood by the receiver's instances have access to a symbol named 'name', evaluate aBlock

whichClassIncludesSelector: selector
Answer which class in the receiver's hierarchy contains the implementation of selector used by instances of the class (nil if none does)

whichSelectorsAccess: instVarName
Answer a Set of selectors which access the given instance variable

whichSelectorsReferTo: anObject
Returns a Set of selectors that refer to anObject

whichSelectorsReferToByteCode: aByteCode
Return the collection of selectors in the class which reference the byte code, aByteCode

6.9 BlockClosure

Defined in namespace Smalltalk
Category: Language-Implementation
I am a factotum class. My instances represent Smalltalk blocks, portions of executeable code that have access to the environment that they were declared in, take parameters, and can be passed around as objects to be executed by methods outside the current class. Block closures are sent a message to compute their value and create a new execution context; this property can be used in the construction of control flow methods. They also provide some methods that are used in the creation of Processes from blocks.

6.9.1 BlockClosure class: instance creation

block: aCompiledBlock
Answer a BlockClosure that activates the passed CompiledBlock.

numArgs: args numTemps: temps bytecodes: bytecodes depth: depth literals: literalArray
Answer a BlockClosure for a new CompiledBlock that is created using the passed parameters. To make it work, you must put the BlockClosure into a CompiledMethod's literals.

6.9.2 BlockClosure class: testing

isImmediate
Answer whether, if x is an instance of the receiver, x copy == x

6.9.3 BlockClosure: accessing

argumentCount
Answer the number of arguments passed to the receiver

block Answer the CompiledBlock which contains the receiver's bytecodes

block: aCompiledBlock
Set the CompiledBlock which contains the receiver's bytecodes

finalIP Answer the last instruction that can be executed by the receiver

fixTemps This should fix the values of the temporary variables used in the block that are ordinarily shared with the method in which the block is defined. Not defined yet, but it is not harmful that it isn't. Answer the receiver.

initialIP Answer the initial instruction pointer into the receiver.

method Answer the CompiledMethod in which the receiver lies

numArgs Answer the number of arguments passed to the receiver

numTemps
Answer the number of temporary variables used by the receiver

outerContext
Answer the method/block context which is the immediate outer of the receiver

outerContext: containingContext
Set the method/block context which is the immediate outer of the receiver

receiver Answer the object that is used as 'self' when executing the receiver (if nil, it might mean that the receiver is not valid though...)

receiver: anObject
Set the object that is used as 'self' when executing the receiver

stackDepth
Answer the number of stack slots needed for the receiver

6.9.4 BlockClosure: built ins

blockCopy: outerContext
Generate a BlockClosure identical to the receiver, with the given context as its outer context.

value Evaluate the receiver passing no parameters

value: arg1
Evaluate the receiver passing arg1 as the only parameter

value: arg1 value: arg2
Evaluate the receiver passing arg1 and arg2 as the parameters

value: arg1 value: arg2 value: arg3
Evaluate the receiver passing arg1, arg2 and arg3 as the parameters

valueWithArguments: argumentsArray
Evaluate the receiver passing argArray's elements as the parameters

6.9.5 BlockClosure: control structures

repeat Evaluate the receiver 'forever' (actually until a return is executed or the process is terminated).

whileFalse Evaluate the receiver until it returns true

whileFalse: aBlock
Evaluate the receiver. If it returns false, evaluate aBlock and re- start

whileTrue Evaluate the receiver until it returns false

whileTrue: aBlock
Evaluate the receiver. If it returns true, evaluate aBlock and re- start

6.9.6 BlockClosure: exception handling

ensure: aBlock
Evaluate the receiver; when any exception is signaled exit returning the result of evaluating aBlock; if no exception is raised, return the result of evaluating aBlock when the receiver has ended

ifCurtailed: aBlock
Evaluate the receiver; when any exception is signaled exit returning the result of evaluating aBlock; if no exception is raised, return the result of evaluating the receiver

ifError: aBlock
Evaluate the receiver; when #error: is called, pass to aBlock the receiver and the parameter, and answer the result of evaluating aBlock. If another exception is raised, it is passed to an outer handler; if no exception is raised, the result of evaluating the receiver is returned.

on: anException do: aBlock
Evaluate the receiver; when anException is signaled, evaluate aBlock passing a Signal describing the exception. Answer either the result of evaluating the receiver or the parameter of a Signal>>#return:

on: e1 do: b1 on: e2 do: b2
Evaluate the receiver; when e1 or e2 are signaled, evaluate respectively b1 or b2, passing a Signal describing the exception. Answer either the result of evaluating the receiver or the argument of a Signal>>#return:

on: e1 do: b1 on: e2 do: b2 on: e3 do: b3
Evaluate the receiver; when e1, e2 or e3 are signaled, evaluate respectively b1, b2 or b3, passing a Signal describing the exception. Answer either the result of evaluating the receiver or the parameter of a Signal>>#return:

on: e1 do: b1 on: e2 do: b2 on: e3 do: b3 on: e4 do: b4
Evaluate the receiver; when e1, e2, e3 or e4 are signaled, evaluate respectively b1, b2, b3 or b4, passing a Signal describing the exception. Answer either the result of evaluating the receiver or the parameter of a Signal>>#return:

on: e1 do: b1 on: e2 do: b2 on: e3 do: b3 on: e4 do: b4 on: e5 do: b5
Evaluate the receiver; when e1, e2, e3, e4 or e5 are signaled, evaluate respectively b1, b2, b3, b4 or b5, passing a Signal describing the exception. Answer either the result of evaluating the receiver or the parameter of a Signal>>#return:

valueWithUnwind
Evaluate the receiver. Any errors caused by the block will cause a backtrace, but execution will continue in the method that sent #valueWithUnwind, after that call. Example: [1 / 0] valueWithUnwind. 'unwind works!' printNl. Important: this method is public, but it is intended to be used in very special cases. You should usually rely on #ensure: and #on:do:

6.9.7 BlockClosure: multiple process

fork Create a new process executing the receiver and start it

forkAt: priority
Create a new process executing the receiver with given priority and start it

forkWithoutPreemption
Evaluate the receiver in a process that cannot be preempted. If the receiver expect a parameter, pass the current process (can be useful for queuing interrupts from within the uninterruptible process).

newProcess
Create a new process executing the receiver in suspended state. The priority is the same as for the calling process. The receiver must not contain returns

newProcessWith: anArray
Create a new process executing the receiver with the passed arguments, and leave it in suspended state. The priority is the same as for the calling process. The receiver must not contain returns

valueWithoutPreemption
Evaluate the receiver without ever having it pre-empted by another process. This selector name is deprecated; use #forkWithoutPreemption instead.

6.9.8 BlockClosure: overriding

deepCopy Answer the receiver.

shallowCopy
Answer the receiver.

6.9.9 BlockClosure: testing

hasMethodReturn
Answer whether the block contains a method return

6.10 BlockContext

Defined in namespace Smalltalk
Category: Language-Implementation
My instances represent executing Smalltalk blocks, which are portions of executeable code that have access to the environment that they were declared in, take parameters, and result from BlockClosure objects created to be executed by methods outside the current class. Block contexts are created by messages sent to compute a closure's value. They contain a stack and also provide some methods that can be used in inspection or debugging.

6.10.1 BlockContext: accessing

caller Answer the context that called the receiver

home Answer the MethodContext to which the receiver refers, or nil if it has been optimized away

isBlock Answer whether the receiver is a block context

isEnvironment
To create a valid execution environment for the interpreter even before it starts, GST creates a fake context whose selector is nil and which can be used as a marker for the current execution environment. Answer whether the receiver is that kind of context (always false, since those contexts are always MethodContexts).

nthOuterContext: n
Answer the n-th outer block/method context for the receiver

outerContext
Answer the outer block/method context for the receiver

6.10.2 BlockContext: printing

printOn: aStream
Print a representation for the receiver on aStream

6.11 Boolean

Defined in namespace Smalltalk
Category: Language-Data types
I have two instances in the Smalltalk system: true and false. I provide methods that are conditional on boolean values, such as conditional execution and loops, and conditional testing, such as conditional and and conditional or. I should say that I appear to provide those operations; my subclasses True and False actually provide those operations.

6.11.1 Boolean class: testing

isIdentity Answer whether x = y implies x == y for instances of the receiver

isImmediate
Answer whether, if x is an instance of the receiver, x copy == x

6.11.2 Boolean: basic

& aBoolean
This method's functionality should be implemented by subclasses of Boolean

and: aBlock
This method's functionality should be implemented by subclasses of Boolean

eqv: aBoolean
This method's functionality should be implemented by subclasses of Boolean

ifFalse: falseBlock
This method's functionality should be implemented by subclasses of Boolean

ifFalse: falseBlock ifTrue: trueBlock
This method's functionality should be implemented by subclasses of Boolean

ifTrue: trueBlock
This method's functionality should be implemented by subclasses of Boolean

ifTrue: trueBlock ifFalse: falseBlock
This method's functionality should be implemented by subclasses of Boolean

not This method's functionality should be implemented by subclasses of Boolean

or: aBlock This method's functionality should be implemented by subclasses of Boolean

xor: aBoolean
This method's functionality should be implemented by subclasses of Boolean

| aBoolean
This method's functionality should be implemented by subclasses of Boolean

6.11.3 Boolean: C hacks

asCBooleanValue
This method's functionality should be implemented by subclasses of Boolean

6.11.4 Boolean: overriding

deepCopy Answer the receiver.

shallowCopy
Answer the receiver.

6.11.5 Boolean: storing

storeOn: aStream
Store on aStream some Smalltalk code which compiles to the receiver

6.12 Browser

Defined in namespace Smalltalk
Category: Language-Implementation

6.12.1 Browser class: browsing

browseHierarchy
Tell Emacs tp browse the Smalltalk class hierarchy

browseMethods: methods forClass: class inBuffer: bufferName
Send to Emacs code that browses the methods in the 'methods' Dictionary, showing them as part of the 'class' class in a buffer with the given name

emacsFunction: funcName on: aBlock
Send to Emacs something like (funcName <aBlock is evaluated here>)

emacsListFunction: funcName on: aBlock
Send to Emacs something like (funcName '(<aBlock is evaluated here>))

finishEmacsMessage
Finish a message to be processed by emacs - does nothing for now

getAllSelectors: selector inBuffer: bufferName
Send to Emacs code that browses the implementors of the given selectors in a buffer with the given name

initialize Initialize the Emacs browsing system

loadClassNames
Tell Emacs the class names (new version)

oldloadClassNames
Tell Emacs the class names

oldShowInstanceMethods: class
Send to Emacs code that browses instance methods for class

oldShowMethods: class for: methodType
Send to Emacs code that browses methods of the given type for class (methodType is either 'class' or 'instance')

selectorsForEmacs
Tell Emacs the names of ALL the defined selectors

showAllMethods: class inBuffer: bufferName
Send to Emacs code that browses ALL the methods understood by instances of the given class, in a buffer with the given name

showDirectMethods: class inBuffer: bufferName
Send to Emacs code that browses methods defined in the given class, in a buffer with the given name

showIndirectMethods: class inBuffer: bufferName
Send to Emacs code that browses the methods inherited (and not overridden) by the given class, in a buffer with the given name

showMethods: class for: methodType
Send to Emacs code that browses methods of the given type for class (methodType is either 'class' or 'instance')

startEmacsMessage
Start a message to be processed by emacs as Lisp

testMethods: aClass for: methodType
Send to Emacs code that browses methods of the given type for class (methodType is either 'class' or 'instance')

withGcOff: aBlock
Evaluate aBlock while the 'GC flipping...' message is off

6.13 ByteArray

Defined in namespace Smalltalk
Category: Collections-Unordered
My instances are similar to strings in that they are both represented as a sequence of bytes, but my individual elements are integers, where as a String's elements are characters.

6.13.1 ByteArray: built ins

asCData: aCType
Convert the receiver to a CObject with the given type

byteAt: index
Answer the index-th indexed instance variable of the receiver

byteAt: index put: value
Store the 'value' byte in the index-th indexed instance variable of the receiver

hash Answer an hash value for the receiver

primReplaceFrom: start to: stop with: aByteArray startingAt: srcIndex
Private - Replace the characters from start to stop with the ASCII codes contained in aString (which, actually, can be any variable byte class), starting at the srcIndex location of aString

replaceFrom: start to: stop withString: aString startingAt: srcIndex
Replace the characters from start to stop with the ASCII codes contained in aString (which, actually, can be any variable byte class), starting at the srcIndex location of aString

6.13.2 ByteArray: converting

asString Answer a String whose character's ASCII codes are the receiver's contents

6.13.3 ByteArray: copying

deepCopy Answer a shallow copy of the receiver

shallowCopy
Answer a shallow copy of the receiver

6.13.4 ByteArray: more advanced accessing

charAt: index
Access the C char at the given index in the receiver. The value is returned as a Smalltalk Character. Indices are 1-based just like for other Smalltalk access.

charAt: index put: value
Store as a C char the Smalltalk Character or Integer object identified by 'value', at the given index in the receiver, using sizeof(char) bytes - i.e. 1 byte. Indices are 1-based just like for other Smalltalk access.

doubleAt: index
Access the C double at the given index in the receiver. Indices are 1-based just like for other Smalltalk access.

doubleAt: index put: value
Store the Smalltalk Float object identified by 'value', at the given index in the receiver, writing it like a C double. Indices are 1-based just like for other Smalltalk access.

floatAt: index
Access the C float at the given index in the receiver. Indices are 1-based just like for other Smalltalk access.

floatAt: index put: value
Store the Smalltalk Float object identified by 'value', at the given index in the receiver, writing it like a C float. Indices are 1-based just like for other Smalltalk access.

intAt: index
Access the C int at the given index in the receiver. Indices are 1-based just like for other Smalltalk access.

intAt: index put: value
Store the Smalltalk Integer object identified by 'value', at the given index in the receiver, using sizeof(int) bytes. Indices are 1-based just like for other Smalltalk access.

longAt: index
Access the C long int at the given index in the receiver. Indices are 1-based just like for other Smalltalk access.

longAt: index put: value
Store the Smalltalk Integer object identified by 'value', at the given index in the receiver, using sizeof(long) bytes. Indices are 1-based just like for other Smalltalk access.

objectAt: index
Access the Smalltalk object (OOP) at the given index in the receiver. Indices are 1-based just like for other Smalltalk access.

objectAt: index put: value
Store a pointer (OOP) to the Smalltalk object identified by 'value', at the given index in the receiver. Indices are 1-based just like for other Smalltalk access.

shortAt: index
Access the C short int at the given index in the receiver. Indices are 1-based just like for other Smalltalk access.

shortAt: index put: value
Store the Smalltalk Integer object identified by 'value', at the given index in the receiver, using sizeof(short) bytes. Indices are 1-based just like for other Smalltalk access.

stringAt: index
Access the string pointed by the C 'char *' at the given index in the receiver. Indices are 1-based just like for other Smalltalk access.

stringAt: index put: value
Store the Smalltalk String object identified by 'value', at the given index in the receiver, writing it like a *FRESHLY ALLOCATED* C string. It is the caller's responsibility to free it if necessary. Indices are 1-based just like for other Smalltalk access.

ucharAt: index
Access the C unsigned char at the given index in the receiver. The value is returned as a Smalltalk Character. Indices are 1-based just like for other Smalltalk access.

ucharAt: index put: value
Store as a C char the Smalltalk Character or Integer object identified by 'value', at the given index in the receiver, using sizeof(char) bytes - i.e. 1 byte. Indices are 1-based just like for other Smalltalk access.

uintAt: index
Access the C unsigned int at the given index in the receiver. Indices are 1-based just like for other Smalltalk access.

uintAt: index put: value
Store the Smalltalk Integer object identified by 'value', at the given index in the receiver, using sizeof(int) bytes. Indices are 1-based just like for other Smalltalk access.

ulongAt: index
Access the C unsigned long int at the given index in the receiver. Indices are 1-based just like for other Smalltalk access.

ulongAt: index put: value
Store the Smalltalk Integer object identified by 'value', at the given index in the receiver, using sizeof(long) bytes. Indices are 1-based just like for other Smalltalk access.

unsignedCharAt: index
: Access the C unsigned char at the given index in the receiver. The value is returned as a Smalltalk Character. Indices are 1-based just like for other Smalltalk access.

unsignedCharAt: index put: value
: Store as a C char the Smalltalk Character or Integer object identified by 'value', at the given index in the receiver, using sizeof(char) bytes - i.e. 1 byte. Indices are 1-based just like for other Smalltalk access.

unsignedIntAt: index
: Access the C unsigned int at the given index in the receiver. Indices are 1-based just like for other Smalltalk access.

unsignedIntAt: index put: value
: Store the Smalltalk Integer object identified by 'value', at the given index in the receiver, using sizeof(int) bytes. Indices are 1-based just like for other Smalltalk access.

unsignedLongAt: index
: Access the C unsigned long int at the given index in the receiver. Indices are 1-based just like for other Smalltalk access.

unsignedLongAt: index put: value
: Store the Smalltalk Integer object identified by 'value', at the given index in the receiver, using sizeof(long) bytes. Indices are 1-based just like for other Smalltalk access.

unsignedShortAt: index
: Access the C unsigned short int at the given index in the receiver. Indices are 1-based just like for other Smalltalk access.

unsignedShortAt: index put: value
: Store the Smalltalk Integer object identified by 'value', at the given index in the receiver, using sizeof(short) bytes. Indices are 1-based just like for other Smalltalk access.

ushortAt: index
: Access the C unsigned short int at the given index in the receiver. Indices are 1-based just like for other Smalltalk access.

ushortAt: index put: value
: Store the Smalltalk Integer object identified by 'value', at the given index in the receiver, using sizeof(short) bytes. Indices are 1-based just like for other Smalltalk access.

6.14 ByteStream

Defined in namespace Smalltalk
Category: Streams-Collections
: My instances are read/write streams specially crafted for ByteArrays. They are able to write binary data to them.

6.14.1 ByteStream: basic

next Return the next *character* in the ByteArray

nextByte Return the next byte in the byte array

nextByteArray: numBytes
Return the next numBytes bytes in the byte array

nextLong Return the next 4 bytes in the byte array, interpreted as a 32 bit signed int

nextPut: aChar
Store aChar on the byte array

nextPutAll: aCollection
Write all the objects in aCollection to the receiver

nextPutByte: anInteger
Store anInteger (range: -128..255) on the byte array

nextPutByteArray: aByteArray
Store aByteArray on the byte array

nextPutLong: anInteger
Store anInteger (range: -2^31..2^32-1) on the byte array as 4 bytes

nextPutShort: anInteger
Store anInteger (range: -32768..65535) on the byte array as 2 bytes

nextShort Return the next 2 bytes in the byte array, interpreted as a 16 bit signed int

nextSignedByte
Return the next byte in the byte array, interpreted as a 8 bit signed number

nextUlong Return the next 4 bytes in the byte array, interpreted as a 32 bit unsigned int

nextUshort
Return the next 2 bytes in the byte array, interpreted as a 16 bit unsigned int

6.15 CAggregate

Defined in namespace Smalltalk
Category: Language-C interface

6.15.1 CAggregate class: accessing

alignof Answer the receiver's instances required aligment

sizeof Answer the receiver's instances size

6.15.2 CAggregate: accessing

+ anInteger
: Return another instance of the receiver's class which points at &receiver[anInteger] (or, if you prefer, what 'receiver + anInteger' does in C).

- intOrPtr
: If intOrPtr is an integer, return another instance of the receiver's class pointing at &receiver[-anInteger] (or, if you prefer, what 'receiver - anInteger' does in C). If it is the same class as the receiver, return the difference in chars, i.e. in bytes, between the two pointed addresses (or, if you prefer, what 'receiver - anotherCharPtr' does in C)

addressAt: anIndex
: Access the array, returning a new Smalltalk CObject of the element type, corresponding to the given indexed element of the array. anIndex is zero-based, just like with all other C-style accessing.

decr
: Adjust the pointer by sizeof(elementType) bytes down (i.e. –receiver)

decrBy: anInteger
: Adjust the pointer by anInteger elements down (i.e. receiver -= anInteger)

deref
: Access the object, returning a new Smalltalk object of the element type, corresponding to the dereferenced pointer or to the first element of the array.

deref: aValue
: Modify the object, storing the object of the element type into the pointed address or in the first element of the array.

derefAt: anIndex
: Access the array, returning a new Smalltalk object of the element type, corresponding to the given indexed element of the array. anIndex is zero-based, just like with all other C-style accessing.

derefAt: anIndex put: aValue
: Store in the array the passed Smalltalk object 'aValue', which should be of the element type, corresponding to the given indexed element. anIndex is zero-based, just like with all other C-style accessing.

incr
: Adjust the pointer by sizeof(elementType) bytes up (i.e. ++receiver)

incrBy: anInteger
: Adjust the pointer by anInteger elements up (i.e. receiver += anInteger)

value
: Answer the address of the beginning of the data pointed to by the receiver.

value: aValue
: Set the address of the beginning of the data pointed to by the receiver.

6.16 CArray

Defined in namespace Smalltalk
Category: Language-C interface

6.16.1 CArray: accessing

alignof Answer the receiver's required aligment

sizeof Answer the receiver's size

6.17 CArrayCType

Defined in namespace Smalltalk
Category: Language-C interface

6.17.1 CArrayCType class: instance creation

elementType: aCType
This method should not be called for instances of this class.

elementType: aCType numberOfElements: anInteger
Answer a new instance of CPtrCType that maps an array whose elements are of the given CType, and whose size is exactly anInteger elements (of course, anInteger only matters for allocation, not for access, since no out-of-bounds protection is provided for C objects).

6.17.2 CArrayCType: accessing

alignof Answer the alignment of the receiver's instances

numberOfElements
Answer the number of elements in the receiver's instances

sizeof Answer the size of the receiver's instances

6.18 CBoolean

Defined in namespace Smalltalk
Category: Language-C interface
I return true if a byte is not zero, false otherwise.

6.18.1 CBoolean: accessing

value Get the receiver's value - answer true if it is != 0, false if it is 0.

value: aBoolean
Set the receiver's value - it's the same as for CBytes, but we get a Boolean, not a Character

6.19 CByte

Defined in namespace Smalltalk
Category: Language-C interface

You're a marine. You adapt – you improvise – you overcome
- Gunnery Sgt. Thomas Highway Heartbreak Ridge

6.19.1 CByte class: conversion

scalarIndex
Nothing special in the default case - answer a CType for the receiver

type Nothing special in the default case - answer a CType for the receiver

6.19.2 CByte: accessing

scalarIndex
Nothing special in the default case - answer the receiver's CType

type Answer a CType for the receiver

value Answer the value the receiver is pointing to. The returned is a SmallInteger

value: aValue
Set the receiver to point to the value, aValue (a SmallInteger).

6.20 CChar

Defined in namespace Smalltalk
Category: Language-C interface

6.20.1 CChar class: accessing

alignof Answer the receiver's instances required aligment

scalarIndex
Private - Answer an index referring to the receiver's instances scalar type

sizeof Answer the receiver's instances size

6.20.2 CChar: accessing

alignof Answer the receiver's required aligment

scalarIndex
Private - Answer an index referring to the receiver's scalar type

sizeof Answer the receiver's size

6.21 CCompound

Defined in namespace Smalltalk
Category: Language-C interface

6.21.1 CCompound class: instance creation

new Allocate a new instance of the receiver. To free the memory after GC, remember to call #addToBeFinalized.

type Answer a CType for the receiver

6.21.2 CCompound class: subclass creation

alignof Answer 1, the alignment of an empty struct

compileDeclaration: array
This method's functionality should be implemented by subclasses of CCompound

compileDeclaration: array inject: startOffset into: aBlock
Compile methods that implement the declaration in array. To compute the offset after each field, the value of the old offset plus the new field's size is passed to aBlock, together with the new field's alignment requirements.

compileSize: size align: alignment
Private - Compile sizeof and alignof methods

computeAggregateType: type block: aBlock
Private - Called by computeTypeString:block: for pointers/arrays. Format of type: (array int 3) or (ptr FooStruct)

computeArrayType: type block: aBlock
Private - Called by computeAggregateType:block: for arrays

computePtrType: type block: aBlock
Private - Called by computeAggregateType:block: for pointers

computeTypeString: type block: aBlock
Private - Pass the size, alignment, and description of CType for aBlock, given the field description in 'type' (the second element of each pair).

emitInspectTo: str for: name
Private - Emit onto the given stream the code for adding the given selector to the CCompound's inspector.

initialize Initialize the receiver's TypeMap

newStruct: structName declaration: array
The old way to create a CStruct. Superseded by #subclass:declaration:...

sizeof Answer 0, the size of an empty struct

subclass: structName declaration: array
classVariableNames: cvn poolDictionaries: pd category: category Create a new class with the given name that contains code to implement the given C struct. All the parameters except 'array' are the same as for a standard class creation message; see documentation for more information

6.21.3 CCompound: instance creation

inspect Inspect the contents of the receiver

inspectSelectorList
Answer a list of selectors whose return values should be inspected by #inspect.

6.22 CDouble

Defined in namespace Smalltalk
Category: Language-C interface

6.22.1 CDouble class: accessing

alignof Answer the receiver's instances required aligment

scalarIndex
Private - Answer an index referring to the receiver's instances scalar type

sizeof Answer the receiver's instances size

6.22.2 CDouble: accessing

alignof Answer the receiver's required aligment

scalarIndex
Private - Answer an index referring to the receiver's scalar type

sizeof Answer the receiver's size

6.23 CFloat

Defined in namespace Smalltalk
Category: Language-C interface

6.23.1 CFloat class: accessing

alignof Answer the receiver's instances required aligment

scalarIndex
Private - Answer an index referring to the receiver's instances scalar type

sizeof Answer the receiver's instances size

6.23.2 CFloat: accessing

alignof Answer the receiver's required aligment

scalarIndex
Private - Answer an index referring to the receiver's scalar type

sizeof Answer the receiver's size

6.24 CFunctionDescriptor

Defined in namespace Smalltalk
Category: Language-C interface
I am not part of the Smalltalk definition. My instances contain information about C functions that can be called from within Smalltalk, such as number and type of parameters. This information is used by the C callout mechanism to perform the actual call-out to C routines.

6.24.1 CFunctionDescriptor class: testing

addressOf: function
Answer the address (CObject) of the function which is registered (on the C side) with the given name, or zero if no such a function is registered.

isFunction: function
Answer whether a function is registered (on the C side) with the given name or is dynamically loadable.

6.24.2 CFunctionDescriptor: accessing

address Answer the address (CObject) of the function represented by the receiver

address: aCObject
Set to aCObject the address of the function represented by the receiver

isValid Answer whether the function represented by the receiver is actually a registered one

name Answer the name of the function (on the C side) represented by the receiver

6.24.3 CFunctionDescriptor: printing

printOn: aStream
Print a representation of the receiver onto aStream

6.25 Character

Defined in namespace Smalltalk
Category: Language-Data types

My instances represent the 256 characters of the character set. I provide messages to translate between integers and character objects, and provide names for some of the common unprintable characters.

6.25.1 Character class: built ins

asciiValue: anInteger

Returns the character object corresponding to anInteger. Error if anInteger is not an integer, or not in 0..255. #codePoint:, #asciiValue: and #value: are synonyms.

codePoint: anInteger

Returns the character object corresponding to anInteger. Error if anInteger is not an integer, or not in 0..255. #codePoint:, #asciiValue: and #value: are synonyms.

value: anInteger

Returns the character object corresponding to anInteger. Error if anInteger is not an integer, or not in 0..255. #codePoint:, #asciiValue: and #value: are synonyms.

6.25.2 Character class: constants

backspace Returns the character 'backspace'

bell Returns the character 'bel'

cr Returns the character 'cr'

eof Returns the character 'eof', also known as 'sub'

eot Returns the character 'eot', also known as 'Ctrl-D'

esc Returns the character 'esc'

lf Returns the character 'lf', also known as 'nl'

newPage Returns the character 'newPage', also known as 'ff'

nl Returns the character 'nl', also known as 'lf'

nul Returns the character 'nul'

space Returns the character 'space'

tab Returns the character 'tab'

6.25.3 Character class: initializing lookup tables

initialize Initialize the lookup table which is used to make case and digit-to-char conversions faster. Indices in Table are ASCII values incremented by one. Indices 1-256 classify chars (0 = nothing special, 2 = separator, 48 = digit, 55 = uppercase, 3 = lowercase), indices 257-512 map to lowercase chars, indices 513-768 map to uppercase chars.

6.25.4 Character class: Instance creation

digitValue: anInteger
Returns a character that corresponds to anInteger. 0-9 map to $0-$9, 10-35 map to $A-$Z

6.25.5 Character class: testing

isIdentity Answer whether x = y implies x == y for instances of the receiver

isImmediate
Answer whether, if x is an instance of the receiver, x copy == x

6.25.6 Character: built ins

= char Boolean return value; true if the characters are equal

asciiValue Returns the integer value corresponding to self. #codePoint, #asciiValue, -#value, and #asInteger are synonyms.

asInteger Returns the integer value corresponding to self. #codePoint, #asciiValue, -#value, and #asInteger are synonyms.

codePoint Returns the integer value corresponding to self. #codePoint, #asciiValue, -#value, and #asInteger are synonyms.

value Returns the integer value corresponding to self. #codePoint, #asciiValue, -#value, and #asInteger are synonyms.

6.25.7 Character: Coercion methods

asLowercase
Returns self as a lowercase character if it's an uppercase letter, otherwise returns the character unchanged.

asString Returns the character self as a string.

asSymbol Returns the character self as a symbol.

asUppercase
Returns self as a uppercase character if it's an lowercase letter, otherwise returns the character unchanged.

6.25.8 Character: comparing

< aCharacter
: Compare the character's ASCII value. Answer whether the receiver's is the least.

<= aCharacter
: Compare the character's ASCII value. Answer whether the receiver's is the least or their equal.

> aCharacter
: Compare the character's ASCII value. Answer whether the receiver's is the greatest.

>= aCharacter
: Compare the character's ASCII value. Answer whether the receiver's is the greatest or their equal.

6.25.9 Character: converting

digitValue Returns the value of self interpreted as a digit. Here, 'digit' means either 0-9, or A-Z, which maps to 10-35.

6.25.10 Character: printing

displayOn: aStream
: Print a representation of the receiver on aStream. Unlike #printOn:, this method strips the leading dollar.

printOn: aStream
: Store a representation of the receiver on aStream

6.25.11 Character: storing

storeOn: aStream
: Store Smalltalk code compiling to the receiver on aStream

6.25.12 Character: testing

isAlphaNumeric
: True if self is a letter or a digit

isDigit True if self is a 0-9 digit

isLetter True if self is an upper- or lowercase letter

isLowercase
: True if self is a lowercase letter

isPunctuation
Returns true if self is one of '.,:;!?'

isSeparator
Returns true if self is a space, cr, tab, nl, or newPage

isUppercase
True if self is uppercase

isVowel Returns true if self is a, e, i, o, or u; case insensitive

6.25.13 Character: testing functionality

isCharacter
Answer True. We're definitely characters

6.26 CharacterArray

Defined in namespace Smalltalk
Category: Language-Data types
My instances represent a generic textual (string) data type. I provide accessing and manipulation methods for strings.

6.26.1 CharacterArray class: basic

fromString: aCharacterArray
Make up an instance of the receiver containing the same characters as aCharacterArray, and answer it.

lineDelimiter
Answer a CharacterArray which one can use as a line delimiter.

6.26.2 CharacterArray: basic

basicAt: index
Answer the index-th character of the receiver. This is an exception to the 'do not override' rule that allows storage optimization by storing the characters as values instead of as objects.

basicAt: index put: anObject
Set the index-th character of the receiver to be anObject. This method must not be overridden; override at: instead. String overrides it so that it looks like it contains character objects even though it contains bytes

6.26.3 CharacterArray: built ins

valueAt: index
Answer the ascii value of index-th character variable of the receiver

valueAt: index put: value
Store (Character value: value) in the index-th indexed instance variable of the receiver

6.26.4 CharacterArray: comparing

< aCharacterArray
Return true if the receiver is less than aCharacterArray, ignoring case differences.

<= aCharacterArray
Returns true if the receiver is less than or equal to aCharacterArray, ignoring case differences. If is receiver is an initial substring of aCharacterArray, it is considered to be less than aCharacterArray.

> aCharacterArray
Return true if the receiver is greater than aCharacterArray, ignoring case differences.

>= aCharacterArray
Returns true if the receiver is greater than or equal to aCharacterArray, ignoring case differences. If is aCharacterArray is an initial substring of the receiver, it is considered to be less than the receiver.

indexOf: aCharacterArray matchCase: aBoolean startingAt: anIndex
Answer an Interval of indices in the receiver which match the aCharacterArray pattern. # in aCharacterArray means 'match any character', * in aCharacterArray means 'match any sequence of characters'. The first item of the returned in- terval is >= anIndex. If aBoolean is false, the search is case-insen- sitive, else it is case-sensitive. If no Interval matches the pattern, answer nil.

match: aCharacterArray
Answer whether aCharacterArray matches the pattern contained in the receiver. # in the receiver means 'match any character', * in receiver means 'match any sequence of characters'.

sameAs: aCharacterArray
Returns true if the receiver is the same CharacterArray as aCharacterArray, ignoring case differences.

6.26.5 CharacterArray: converting

asByteArray
Return the receiver, converted to a ByteArray of ASCII values

asClassPoolKey
Return the receiver, ready to be put in a class pool dictionary

asGlobalKey
Return the receiver, ready to be put in the Smalltalk dictionary

asInteger Parse an Integer number from the receiver until the input character is invalid and answer the result at this point

asLowercase
Returns a copy of self as a lowercase CharacterArray

asNumber Parse a Number from the receiver until the input character is invalid and answer the result at this point

asPoolKey Return the receiver, ready to be put in a pool dictionary

asString But I already am a String! Really!

asSymbol Returns the symbol corresponding to the CharacterArray

asUppercase
Returns a copy of self as an uppercase CharacterArray

fileName But I don't HAVE a file name!

filePos But I don't HAVE a file position!

isNumeric Answer whether the receiver denotes a number

trimSeparators
Return a copy of the reciever without any spaces on front or back. The implementation is protected against the 'all blanks' case.

6.26.6 CharacterArray: copying

deepCopy Returns a deep copy of the receiver. This is the same thing as a shallow copy for CharacterArrays

shallowCopy
Returns a shallow copy of the receiver

6.26.7 CharacterArray: printing

displayOn: aStream
Print a representation of the receiver on aStream. Unlike #printOn:, this method strips extra quotes.

displayString
Answer a String representing the receiver. For most objects this is simply its #printString, but for CharacterArrays and characters, superfluous dollars or extra pair of quotes are stripped.

printOn: aStream
Print a representation of the receiver on aStream

6.26.8 CharacterArray: storing

storeOn: aStream
Print Smalltalk code compiling to the receiver on aStream

6.26.9 CharacterArray: string processing

bindWith: s1
Answer the receiver with every %1 replaced by the displayString of s1

bindWith: s1 with: s2
Answer the receiver with every %1 or %2 replaced by s1 or s2, respectively. s1 and s2 are 'displayed' (i.e. their displayString is used) upon replacement.

bindWith: s1 with: s2 with: s3
Answer the receiver with every %1, %2 or %3 replaced by s1, s2 or s3, respectively. s1, s2 and s3 are 'displayed' (i.e. their displayString is used) upon replacement.

bindWith: s1 with: s2 with: s3 with: s4
Answer the receiver with every %1, %2, %3 or %4 replaced by s1, s2, s3 or s4, respectively. s1, s2, s3 and s4 are 'displayed' (i.e. their displayString is used) upon replacement.

bindWithArguments: anArray
Answer the receiver with every %n (1<=n<=9) replaced by the n-th element of anArray. The replaced elements are 'displayed' (i.e. their displayString is used)

contractTo: smallSize
Either return myself, or a copy shortened to smallSize characters by inserting an ellipsis (three dots: ...)

substrings Answer an OrderedCollection of substrings of the receiver. A new substring start at the start of the receiver, or after every sequence of white space characters

substrings: aCharacter
Answer an OrderedCollection of substrings of the receiver. A new substring start at the start of the receiver, or after every sequence of characters matching aCharacter. This message is preserved for backwards compatibility; the ANSI standard mandates 'subStrings:', with an uppercase s.

subStrings: aCharacter
Answer an OrderedCollection of substrings of the receiver. A new substring start at the start of the receiver, or after every sequence of characters matching aCharacter

6.26.10 CharacterArray: testing functionality

isCharacterArray
Answer 'true'.

6.27 CInt

Defined in namespace Smalltalk
Category: Language-C interface

6.27.1 CInt class: accessing

alignof Answer the receiver's required aligment

scalarIndex
Private - Answer an index referring to the receiver's instances scalar type

sizeof Answer the receiver's size

6.27.2 CInt: accessing

alignof Answer the receiver's instances required aligment

scalarIndex
Private - Answer an index referring to the receiver's scalar type

sizeof Answer the receiver's instances size

6.28 Class

Defined in namespace Smalltalk
Category: Language-Implementation
I am THE class object. My instances are the classes of the system. I provide information commonly attributed to classes: namely, the class name, class comment (you wouldn't be reading this if it weren't for me), a list of the instance variables of the class, and the class category.

6.28.1 Class: accessing instances and variables

addClassVarName: aString
Add a class variable with the given name to the class pool dictionary

addSharedPool: aDictionary
Add the given shared pool to the list of the class' pool dictionaries

allClassVarNames
Answer the names of the variables in the receiver's class pool dictionary and in each of the superclasses' class pool dictionaries

category Answer the class category

category: aString
Change the class category to aString

classPool Answer the class pool dictionary

classVarNames
Answer the names of the variables in the class pool dictionary

comment Answer the class comment

comment: aString
Change the class name

environment
Answer 'environment'.

environment: aNamespace
Set the receiver's environment to aNamespace and recompile everything

initialize redefined in children (?)

name Answer the class name

removeClassVarName: aString
Removes the class variable from the class, error if not present, or still in use.

removeSharedPool: aDictionary
Remove the given dictionary to the list of the class' pool dictionaries

sharedPools
Return the names of the shared pools defined by the class

6.28.2 Class: filing

fileOutDeclarationOn: aFileStream
File out class definition to aFileStream

fileOutHeaderOn: aFileStream
Write date and time stamp to aFileStream

fileOutOn: aFileStream
File out complete class description: class definition, class and instance methods

6.28.3 Class: instance creation

extend Redefine a version of the receiver in the current namespace. Note: this method can bite you in various ways when sent to system classes; read the section on namespaces in the manual for some examples of the problems you can encounter.

subclass: classNameString
instanceVariableNames: stringInstVarNames classVariableNames: stringOfClassVarNames poolDictionaries: stringOfPoolNames category: categoryNameString Define a fixed subclass of the receiver with the given name, instance variables, class variables, pool dictionaries and category. If the class is already defined, if necessary, recompile everything needed.

variableByteSubclass: classNameString
instanceVariableNames: stringInstVarNames classVariableNames: stringOfClassVarNames poolDictionaries: stringOfPoolNames category: categoryNameString Define a byte variable subclass of the receiver with the given name, instance variables (must be ''), class variables, pool dictionaries and category. If the class is already defined, if necessary, recompile everything needed.

variableSubclass: classNameString
instanceVariableNames: stringInstVarNames classVariableNames: stringOfClassVarNames poolDictionaries: stringOfPoolNames category: categoryNameString Define a variable pointer subclass of the receiver with the given name, instance variables, class variables, pool dictionaries and category. If the class is already defined, if necessary, recompile everything needed.

variableWordSubclass: classNameString
instanceVariableNames: stringInstVarNames classVariableNames: stringOfClassVarNames poolDictionaries: stringOfPoolNames category: categoryNameString Define a word variable subclass of the receiver with the given name, instance variables (must be ''), class variables, pool dictionaries and category. If the class is already defined, if necessary, recompile everything needed.

6.28.4 Class: instance creation - alternative

categoriesFor: method are: categories
Don't use this, it is only present to file in from IBM Smalltalk

subclass: classNameString classInstanceVariableNames: stringClassInstVarNames instanceVariableNames: stringInstVarNames classVariableNames: stringOfClassVarNames poolDictionaries: stringOfPoolNames
Don't use this, it is only present to file in from IBM Smalltalk

subclass: classNameString instanceVariableNames: stringInstVarNames classVariableNames: stringOfClassVarNames poolDictionaries: stringOfPoolNames
Don't use this, it is only present to file in from IBM Smalltalk

variableByteSubclass: classNameString classInstanceVariableNames: stringClassInstVarNames classVariableNames: stringOfClassVarNames poolDictionaries: stringOfPoolNames
Don't use this, it is only present to file in from IBM Smalltalk

variableByteSubclass: classNameString classVariableNames: stringOfClassVarNames poolDictionaries: stringOfPoolNames
Don't use this, it is only present to file in from IBM Smalltalk

variableLongSubclass: classNameString classInstanceVariableNames: stringClassInstVarNames classVariableNames: stringOfClassVarNames poolDictionaries: stringOfPoolNames
Don't use this, it is only present to file in from IBM Smalltalk

variableLongSubclass: classNameString classVariableNames: stringOfClassVarNames poolDictionaries: stringOfPoolNames
Don't use this, it is only present to file in from IBM Smalltalk

variableSubclass: classNameString classInstanceVariableNames: stringClassInstVarNames instanceVariableNames: stringInstVarNames classVariableNames: stringOfClassVarNames poolDictionaries: stringOfPoolNames
Don't use this, it is only present to file in from IBM Smalltalk

variableSubclass: classNameString instanceVariableNames: stringInstVarNames classVariableNames: stringOfClassVarNames poolDictionaries: stringOfPoolNames
Don't use this, it is only present to file in from IBM Smalltalk

6.28.5 Class: printing

article Answer an article ('a' or 'an') which is ok for the receiver's name

printOn: aStream
Print a representation of the receiver on aStream

storeOn: aStream
Store Smalltalk code compiling to the receiver on aStream

6.28.6 Class: saving and loading

binaryRepresentationVersion
Answer a number >= 0 which represents the current version of the object's representation. The default implementation answers zero.

convertFromVersion: version withFixedVariables: fixed
indexedVariables: indexed for: anObjectDumper This method is called if a VersionableObjectProxy is attached to a class. It receives the version number that was stored for the object (or nil if the object did not use a VersionableObjectProxy), the fixed instance variables, the indexed instance variables, and the ObjectDumper that has read the object. The default implementation ignores the version and simply fills in an instance of the receiver with the given fixed and indexed instance variables (nil if the class instances are of fixed size). If instance variables were removed from the class, extras are ignored; if the class is now fixed and used to be indexed, indexed is not used.

nonVersionedInstSize
Answer the number of instance variables that the class used to have when objects were stored without using a VersionableObjectProxy. The default implementation answers the current instSize.

6.28.7 Class: testing

= aClass Returns true if the two class objects are to be considered equal.

6.28.8 Class: testing functionality

asClass Answer the receiver.

isClass Answer 'true'.

6.29 ClassDescription

Defined in namespace Smalltalk
Category: Language-Implementation
My instances provide methods that access classes by category, and allow whole categories of classes to be filed out to external disk files.

6.29.1 ClassDescription: compiling

compile: code classified: categoryName
Compile code in the receiver, assigning the method to the given category. Answer the newly created CompiledMethod, or nil if an error was found.

compile: code classified: categoryName ifError: block
Compile method source and install in method category, categoryName. If there are parsing errors, invoke exception block, 'block' (see compile:ifError:). Return the method

compile: code classified: categoryName notifying: requestor
Compile method source and install in method category, categoryName. If there are parsing errors, send an error message to requestor

6.29.2 ClassDescription: conversion

asClass This method's functionality should be implemented by subclasses of ClassDescription

asMetaclass
Answer the metaclass associated to the receiver

6.29.3 ClassDescription: copying

copy: selector from: aClass
Copy the given selector from aClass, assigning it the same category

copy: selector from: aClass classified: categoryName
Copy the given selector from aClass, assigning it the given category

copyAll: arrayOfSelectors from: class
Copy all the selectors in arrayOfSelectors from class, assigning them the same category they have in class

copyAll: arrayOfSelectors from: class classified: categoryName
Copy all the selectors in arrayOfSelectors from aClass, assigning them the given category

copyAllCategoriesFrom: aClass
Copy all the selectors in aClass, assigning them the original category

copyCategory: categoryName from: aClass
Copy all the selectors in from aClass that belong to the given category

copyCategory: categoryName from: aClass classified: newCategoryName
Copy all the selectors in from aClass that belong to the given category, reclassifying them as belonging to the given category

6.29.4 ClassDescription: filing

fileOut: fileName
Open the given file and to file out a complete class description to it

fileOutCategory: categoryName to: fileName
File out all the methods belonging to the method category, categoryName, to the fileName file

fileOutCategory: category toStream: aFileStream
File out all the methods belonging to the method category, categoryName, to aFileStream

fileOutOn: aFileStream
File out complete class description: class definition, class and instance methods

fileOutSelector: selector to: fileName
File out the given selector to fileName

6.29.5 ClassDescription: organization of messages and classes

createGetMethod: what
Create a method accessing the variable 'what'.

createGetMethod: what default: value
Create a method accessing the variable 'what', with a default value of 'value', using lazy initialization

createSetMethod: what
Create a method which sets the variable 'what'.

defineCFunc: cFuncNameString
withSelectorArgs: selectorAndArgs returning: returnTypeSymbol args: argsArray See documentation. Too complex to describe it here ;-)

removeCategory: aString
Remove from the receiver every method belonging to the given category

whichCategoryIncludesSelector: selector
Answer the category for the given selector, or nil if the selector is not found

6.29.6 ClassDescription: printing

classVariableString
This method's functionality should be implemented by subclasses of ClassDescription

instanceVariableString
Answer a string containing the name of the receiver's instance variables.

nameIn: aNamespace
Answer the class name when the class is referenced from aNamespace

sharedVariableString
This method's functionality should be implemented by subclasses of ClassDescription

6.30 CLong

Defined in namespace Smalltalk
Category: Language-C interface

6.30.1 CLong class: accessing

alignof Answer the receiver's instances required aligment

scalarIndex
Private - Answer an index referring to the receiver's instances scalar type

sizeof Answer the receiver's instances size

6.30.2 CLong: accessing

alignof Answer the receiver's required aligment

scalarIndex
Private - Answer an index referring to the receiver's scalar type

sizeof Answer the receiver's size

6.31 CObject

Defined in namespace Smalltalk
Category: Language-C interface
I am not part of the standard Smalltalk kernel class hierarchy. My instances contain values that are not interpreted by the Smalltalk system; they frequently hold "pointers" to data outside of the Smalltalk environment. The C callout mechanism allows my instances to be transformed into their corresponding C values for use in external routines.

6.31.1 CObject class: conversion

scalarIndex
Nothing special in the default case - answer a CType for the receiver

type Nothing special in the default case - answer a CType for the receiver

6.31.2 CObject class: instance creation

address: anInteger
Answer a new object pointing to the passed address, anInteger

alloc: nBytes
Allocate nBytes bytes and return an instance of the receiver

alloc: nBytes type: cTypeObject
Allocate nBytes bytes and return a CObject of the given type

new: nBytes
Allocate nBytes bytes and return an instance of the receiver

6.31.3 CObject: accessing

address Answer the address the receiver is pointing to.

address: anInteger
Set the receiver to point to the passed address, anInteger

at: byteOffset
Answer some data of the receiver's default type, reading byteOffset bytes after the pointer stored in the receiver

at: byteOffset put: value
Store some data of the receiver's default type, writing byteOffset bytes after the pointer stored in the receiver

printOn: aStream
Print a representation of the receiver

type: aCType
Set the receiver's type to aCType.

value What can I return? So fail

value: anObject
What can I set? So fail

6.31.4 CObject: C data access

at: byteOffset put: aValue type: aType
Store aValue as data of the given type from byteOffset bytes after the pointer stored in the receiver

at: byteOffset type: aType
Answer some data of the given type from byteOffset bytes after the pointer stored in the receiver

free Free the receiver's pointer and set it to null. Big trouble hits you if the receiver doesn't point to the base of a malloc-ed area.

6.31.5 CObject: conversion

castTo: aType
Answer another CObject, pointing to the same address as the receiver, but belonging to the aType CType.

scalarIndex
Nothing special in the default case - answer the receiver's CType

type Answer a CType for the receiver

6.31.6 CObject: finalization

finalize To make the VM call this, use #addToBeFinalized. It frees automatically any memory pointed to by the CObject. It is not automatically enabled because big trouble hits you if you use #free and the receiver doesn't point to the base of a malloc-ed area.

6.32 Collection

Defined in namespace Smalltalk
Category: Collections
I am an abstract class. My instances are collections of objects. My subclasses may place some restrictions or add some definitions to how the objects are stored or organized; I say nothing about this. I merely provide some object creation and access routines for general collections of objects.

6.32.1 Collection class: instance creation

with: anObject
Answer a collection whose only element is anObject

with: firstObject with: secondObject
Answer a collection whose only elements are the parameters in the order they were passed

with: firstObject with: secondObject with: thirdObject
Answer a collection whose only elements are the parameters in the order they were passed

with: firstObject with: secondObject with: thirdObject with: fourthObject
Answer a collection whose only elements are the parameters in the order they were passed

with: firstObject with: secondObject with: thirdObject with: fourthObject with: fifthObject
Answer a collection whose only elements are the parameters in the order they were passed

withAll: aCollection
Answer a collection whose elements are all those in aCollection

6.32.2 Collection: Adding to a collection

add: newObject
Add newObject to the receiver, answer it

addAll: aCollection
Adds all the elements of 'aCollection' to the receiver, answer aCollection

6.32.3 Collection: converting

asArray Answer an Array containing all the elements in the receiver

asBag Answer a Bag containing all the elements in the receiver

asByteArray
Answer a ByteArray containing all the elements in the receiver

asOrderedCollection
Answer an OrderedCollection containing all the elements in the receiver

asRunArray
Answer the receiver converted to a RunArray. If the receiver is not ordered the order of the elements in the RunArray might not be the #do: order.

asSet Answer a Set containing all the elements in the receiver with no duplicates

asSortedCollection
Answer a SortedCollection containing all the elements in the receiver with the default sort block - [:a :b | a <= b]

asSortedCollection: aBlock
Answer a SortedCollection whose elements are the elements of the receiver, sorted according to the sort block aBlock

6.32.4 Collection: copying Collections

copyReplacing: targetObject withObject: newObject
Copy replacing each object which is = to targetObject with newObject

copyWith: newElement
Answer a copy of the receiver to which newElement is added

copyWithout: oldElement
Answer a copy of the receiver to which all occurrences of oldElement are removed

6.32.5 Collection: enumerating the elements of a collection

allSatisfy: aBlock
Search the receiver for an element for which aBlock returns false. Answer true if none does, false otherwise.

anyOne Answer an unspecified element of the collection. Example usage: ^coll inject: coll anyOne into: [:max :each | max max: each] to be used when you don't have a valid lowest-possible-value (which happens in common cases too, such as with arbitrary numbers

anySatisfy: aBlock
Search the receiver for an element for which aBlock returns true. Answer true if some does, false otherwise.

beConsistent
This method is private, but it is quite interesting so it is documented. It ensures that a collection is in a consistent state before attempting to iterate on it; its presence reduces the number of overrides needed by collections who try to amortize their execution times. The default implementation does nothing, so it is optimized out by the virtual machine and so it loses very little on the performance side. Note that descendants of Collection have to call it explicitly since #do: is abstract in Collection.

collect: aBlock
Answer a new instance of a Collection containing all the results of evaluating aBlock passing each of the receiver's elements

conform: aBlock
Search the receiver for an element for which aBlock returns false. Answer true if none does, false otherwise.

contains: aBlock
Search the receiver for an element for which aBlock returns true. Answer true if some does, false otherwise.

detect: aBlock
Search the receiver for an element for which aBlock returns true. If some does, answer it. If none does, fail

detect: aBlock ifNone: exceptionBlock
Search the receiver for an element for which aBlock returns true. If some does, answer it. If none does, answer the result of evaluating aBlock

do: aBlock
Enumerate each object of the receiver, passing them to aBlock

do: aBlock separatedBy: separatorBlock
Enumerate each object of the receiver, passing them to aBlock. Between every two invocations of aBlock, invoke separatorBlock

inject: thisValue into: binaryBlock
Pass to binaryBlock receiver thisValue and the first element of the receiver; for each subsequent element, pass the result of the previous evaluation and an element. Answer the result of the last invocation.

reject: aBlock
Answer a new instance of a Collection containing all the elements in the receiver which, when passed to aBlock, don't answer true

select: aBlock
Answer a new instance of a Collection containing all the elements in the receiver which, when passed to aBlock, answer true

6.32.6 Collection: printing

inspect Print all the instance variables and objects in the receiver on the Transcript

printOn: aStream
Print a representation of the receiver on aStream

6.32.7 Collection: Removing from a collection

remove: oldObject
Remove oldObject from the receiver. If absent, fail, else answer oldObject.

remove: oldObject ifAbsent: anExceptionBlock
Remove oldObject from the receiver. If absent, evaluate anExceptionBlock and answer the result, else answer oldObject.

removeAll: aCollection
Remove each object in aCollection, answer aCollection, fail if some of them is absent. Warning: this could leave the collection in a semi-updated state.

removeAll: aCollection ifAbsent: aBlock
Remove each object in aCollection, answer aCollection; if some element is absent, pass it to aBlock.

6.32.8 Collection: storing

storeOn: aStream
Store Smalltalk code compiling to the receiver on aStream

6.32.9 Collection: testing collections

capacity Answer how many elements the receiver can hold before having to grow.

identityIncludes: anObject
Answer whether we include the anObject object

includes: anObject
Answer whether we include anObject

isEmpty Answer whether we are (still) empty

notEmpty Answer whether we include at least one object

occurrencesOf: anObject
Answer how many occurrences of anObject we include

size Answer how many objects we include

6.33 CompiledBlock

Defined in namespace Smalltalk
Category: Language-Implementation
I represent a block that has been compiled.

6.33.1 CompiledBlock class: instance creation

newMethod: numBytecodes header: anInteger method: outerMethod
Answer a new CompiledMethod with room for the given bytes and the given header

numArgs: args numTemps: temps bytecodes: bytecodes depth: depth literals: literalArray
Answer an (almost) full fledged CompiledBlock. To make it complete, you must either set the new object's 'method' variable, or put it into a BlockClosure and put the BlockClosure into a CompiledMethod's literals. The clean-ness of the block is automatically computed.

6.33.2 CompiledBlock: accessing

flags Answer the 'cleanness' of the block. 0 = clean; 1 = access to receiver variables and/or self; 2-30 = access to variables that are 1-29 contexts away; 31 = return from method or push thisContext

method Answer the CompiledMethod in which the receiver lies

methodClass
Answer the class in which the receiver is installed.

methodClass: methodClass
Set the receiver's class instance variable

numArgs Answer the number of arguments passed to the receiver

numLiterals
Answer the number of literals for the receiver

numTemps
Answer the number of temporary variables used by the receiver

selector Answer the selector through which the method is called

selector: aSymbol
Set the selector through which the method is called

stackDepth
Answer the number of stack slots needed for the receiver

6.33.3 CompiledBlock: basic

= aMethod
Answer whether the receiver and aMethod are equal

methodCategory
Answer the method category

methodCategory: aCategory
Set the method category to the given string

methodSourceCode
Answer the method source code (a FileSegment or String or nil)

methodSourceFile
Answer the file where the method source code is stored

methodSourcePos
Answer the location where the method source code is stored in the methodSourceFile

methodSourceString
Answer the method source code as a string

6.33.4 CompiledBlock: printing

printOn: aStream
Print the receiver's class and selector on aStream

6.33.5 CompiledBlock: saving and loading

binaryRepresentationObject
This method is implemented to allow for a PluggableProxy to be used with CompiledBlocks. Answer a DirectedMessage which sends #blockAt: to the CompiledMethod containing the receiver.

6.34 CompiledCode

Defined in namespace Smalltalk
Category: Language-Implementation

I represent code that has been compiled. I am an abstract superclass for blocks and methods

6.34.1 CompiledCode class: cache flushing

flushTranslatorCache

Answer any kind of cache mantained by a just-in-time code translator in the virtual machine (if any). Do nothing for now.

6.34.2 CompiledCode class: instance creation

newMethod: numBytecodes header: anInteger literals: literals

Answer a new CompiledMethod with room for the given bytes and the given header

newMethod: numBytecodes header: anInteger numLiterals: numLiterals

Answer a new CompiledMethod with room for the given bytes and the given header

6.34.3 CompiledCode: accessing

at: anIndex put: aBytecode

Store aBytecode as the anIndex-th bytecode

blockAt: anIndex

Answer the CompiledBlock attached to the anIndex-th literal, assuming that the literal is a BlockClosure.

bytecodeAt: anIndex

Answer the anIndex-th bytecode

bytecodeAt: anIndex put: aBytecode

Store aBytecode as the anIndex-th bytecode

flags Private - Answer the optimization flags for the receiver

literalAt: anIndex

Answer the anIndex-th literal

literalAt: anInteger put: aValue

Store aValue as the anIndex-th literal

literals Answer 'literals'.

methodClass

Answer the class in which the receiver is installed.

methodClass: methodClass
Set the receiver's class instance variable

numArgs Answer the number of arguments for the receiver

numLiterals
Answer the number of literals for the receiver

numTemps
Answer the number of temporaries for the receiver

primitive Answer the primitive called by the receiver

selector Answer the selector through which the method is called

selector: aSymbol
Set the selector through which the method is called

stackDepth
Answer the number of stack slots needed for the receiver

6.34.4 CompiledCode: basic

= aMethod
Answer whether the receiver and aMethod are equal

hash Answer an hash value for the receiver

methodCategory
Answer the method category

methodCategory: aCategory
Set the method category to the given string

methodSourceCode
Answer the method source code (a FileSegment or String or nil)

methodSourceFile
Answer the file where the method source code is stored

methodSourcePos
Answer the location where the method source code is stored in the methodSourceFile

methodSourceString
Answer the method source code as a string

6.34.5 CompiledCode: copying

deepCopy Answer a deep copy of the receiver

6.34.6 CompiledCode: debugging

breakAtLine: lineNumber
This method's functionality has not been implemented yet.

breakpointAt: byteIndex
Put a break-point at the given bytecode

inspect Print the contents of the receiver in a verbose way.

removeBreakpointAt: byteIndex
Remove the break-point at the given bytecode (don't fail if none was set

6.34.7 CompiledCode: printing

printOn: aStream
Print the receiver's class and selector on aStream

6.34.8 CompiledCode: testing accesses

accesses: instVarIndex
Answer whether the receiver access the instance variable with the given index

containsLiteral: anObject
Answer if the receiver contains a literal which is equal to anObject.

hasBytecode: byte between: firstIndex and: lastIndex
Answer whether the receiver includes the 'byte' bytecode in any of the indices between firstIndex and lastIndex.

jumpDestinationAt: anIndex
Answer where the jump at bytecode index 'anIndex' lands

refersTo: anObject
Answer whether the receiver refers to the given object

6.34.9 CompiledCode: translation

discardTranslation
Flush the just-in-time translated code for the receiver (if any).

6.35 CompiledMethod

Defined in namespace Smalltalk
Category: Language-Implementation
I represent methods that have been compiled. I can recompile methods from their source code, I can invoke Emacs to edit the source code for one of my instances, and I know how to access components of my instances.

6.35.1 CompiledMethod class: instance creation

literals: lits numArgs: numArg numTemps: numTemp
primitive: primIndex bytecodes: bytecodes depth: depth Answer a full fledged CompiledMethod. Construct the method header from the parameters, and set the literals and bytecodes to the provided ones. Also, the bytecodes are optimized and any embedded CompiledBlocks modified to refer to these literals and to the newly created CompiledMethod.

6.35.2 CompiledMethod class: lean images

stripSourceCode
Remove all the references to method source code from the system

6.35.3 CompiledMethod: accessing

flags Private - Answer the optimization flags for the receiver

methodClass
Answer the class in which the receiver is installed.

methodClass: methodClass
Set the receiver's class instance variable

numArgs Answer the number of arguments for the receiver

numTemps
Answer the number of temporaries for the receiver

primitive Answer the primitive called by the receiver

selector Answer the selector through which the method is called

selector: aSymbol
Set the selector through which the method is called

stackDepth
Answer the number of stack slots needed for the receiver

withNewMethodClass: class
Answer either the receiver or a copy of it, with the method class set to class

withNewMethodClass: class selector: selector
Answer either the receiver or a copy of it, with the method class set to class

6.35.4 CompiledMethod: basic

= aMethod
Answer whether the receiver and aMethod are equal

hash Answer an hash value for the receiver

methodCategory
Answer the method category

methodCategory: aCategory
Set the method category to the given string

methodSourceCode
Answer the method source code (a FileSegment or String or nil)

methodSourceFile
Answer the file where the method source code is stored

methodSourcePos
Answer the location where the method source code is stored in the methodSourceFile

methodSourceString
Answer the method source code as a string

6.35.5 CompiledMethod: printing

storeOn: aStream
Print code to create the receiver on aStream

6.35.6 CompiledMethod: saving and loading

binaryRepresentationObject
This method is implemented to allow for a PluggableProxy to be used with CompiledMethods. Answer a DirectedMessage which sends #>> to the class object containing the receiver.

6.36 ContextPart

Defined in namespace Smalltalk
Category: Language-Implementation
My instances represent executing Smalltalk code, which represent the local environment of executable code. They contain a stack and also provide some methods that can be used in inspection or debugging.

6.36.1 ContextPart class: exception handling

backtrace Print a backtrace from the caller to the bottom of the stack on the Transcript

backtraceOn: aStream
Print a backtrace from the caller to the bottom of the stack on aStream

lastUnwindPoint
Private - Return the last context marked as an unwind point, or our environment if none is.

removeLastUnwindPoint
Private - Return and remove the last context marked as an unwind point, or our environment if the last unwind point belongs to another environment.

unwind Return execution to the last context marked as an unwind point, returning nil on that stack.

unwind: returnValue
Return execution to the last context marked as an unwind point, returning returnValue on that stack.

6.36.2 ContextPart: accessing

client Answer the client of this context, that is, the object that sent the message that created this context. Fail if the receiver has no parent

environment
To create a valid execution environment for the interpreter even before it starts, GST creates a fake context whose selector is nil and which can be used as a marker for the current execution environment. This method answers that context. For processes, it answers the process block itself

home Answer the MethodContext to which the receiver refers

initialIP Answer the value of the instruction pointer when execution starts in the current context

ip Answer the current instruction pointer into the receiver

ip: newIP Set the instruction pointer for the receiver

isBlock Answer whether the receiver is a block context

isEnvironment
To create a valid execution environment for the interpreter even before it starts, GST creates a fake context whose selector is nil and which can be used as a marker for the current execution environment. Answer whether the receiver is that kind of context.

isProcess Answer whether the receiver represents a process context, i.e. a context created by BlockClosure>>#newProcess. Such a context can be recognized because it has no parent but its flags are different from those of the contexts created by the VM's prepareExecutionEnvironment function.

method Return the CompiledMethod being executed

methodClass
Return the class in which the CompiledMethod being executed is defined

numArgs Answer the number of arguments passed to the receiver

numTemps
Answer the number of temporaries used by the receiver

parentContext
Answer the context that called the receiver

receiver Return the receiver (self) for the method being executed

selector Return the selector for the method being executed

size Answer the number of valid fields for the receiver. Any read access from (self size + 1) to (self basicSize) has undefined results - even crashing

sp Answer the current stack pointer into the receiver

sp: newSP
Set the stack pointer for the receiver.

validSize Answer how many elements in the receiver should be inspected

6.36.3 ContextPart: copying

deepCopy Answer a shallow copy of the receiver – duplicating e.g. the method and the instance variables that have been pushed is almost surely not the right thing.

shallowCopy
Answer a copy of the receiver

6.36.4 ContextPart: enumerating

scanBacktraceFor: selectors do: aBlock
Scan the backtrace for contexts whose selector is among those listed in selectors; if one is found, invoke aBlock passing the selector.

6.36.5 ContextPart: exception handling

mark Add the receiver as a possible unwind point

returnTo: aContext
Set the context to which the receiver will return

6.36.6 ContextPart: printing

backtrace Print a backtrace from the receiver to the bottom of the stack on the Transcript.

backtraceOn: aStream
Print a backtrace from the caller to the bottom of the stack on aStream.

6.37 CoreException

Defined in namespace Smalltalk
Category: Language-Exceptions

My instances describe a single event that can be trapped using #on:do:..., contain whether such execution can be resumed after such an event, a description of what happened, and a block that is used as an handler by default. Using my methods you can raise exceptions and create new exceptions. Exceptions are organized in a kind of hierarchy (different from the class hierarchy): intercepting an exception will intercept all its children too.

CoreExceptions are different from ANSI Exceptions in that the signaled exception is not an instance of the CoreException, instead it belongs to a different class, Signal. ANSI Exceptions inherit from Signal but hold on to a CoreException via a class-instance variable.

6.37.1 CoreException class: instance creation

new Create a new exception whose parent is ExAll

6.37.2 CoreException: accessing

defaultHandler
Answer the default handler for the receiver

defaultHandler: aBlock
Set the default handler of the receiver to aBlock. A Signal object will be passed to aBlock

description
Answer a description of the receiver

description: aString
Set the description of the receiver to aString

isResumable
Answer true if the receiver is resumable

isResumable: aBoolean
Set the resumable flag of the receiver to aBoolean

parent Answer the parent of the receiver

signalClass
Answer the subclass of Signal to be passed to handler blocks that handle the receiver

signalClass: aClass
Set which subclass of Signal is to be passed to handler blocks that handle the receiver

6.37.3 CoreException: basic

copy Answer a copy of the receiver

6.37.4 CoreException: enumerating

allExceptionsDo: aBlock
Private - Evaluate aBlock for every exception in the receiver. As it contains just one exception, evaluate it just once, passing the receiver

handles: exceptionOrSignal
Answer whether the receiver handles 'exceptionOrSignal'.

6.37.5 CoreException: exception handling

signal Raise the exception described by the receiver, passing no parameters

signalWith: arg
Raise the exception described by the receiver, passing the parameter arg

signalWith: arg with: arg2
Raise the exception described by the receiver, passing the parameters arg and arg2

signalWithArguments: args
Raise the exception described by the receiver, passing the parameters in args

6.37.6 CoreException: instance creation

newChild Answer a child exception of the receiver. Its properties are set to those of the receiver

6.38 CPtr

Defined in namespace Smalltalk
Category: Language-C interface

6.38.1 CPtr: accessing

alignof Answer the receiver's required aligment

sizeof Answer the receiver's size

6.39 CPtrCType

Defined in namespace Smalltalk
Category: Language-C interface

6.39.1 CPtrCType class: instance creation

elementType: aCType
Answer a new instance of CPtrCType that maps pointers to the given CType

6.39.2 CPtrCType: accessing

elementType
Answer the type of the elements in the receiver's instances

new: size Allocate space for 'size' objects like those that the receiver points to, and with the type (class) identified by the receiver. It is the caller's responsibility to free the memory allocated for it.

6.40 CScalar

Defined in namespace Smalltalk
Category: Language-C interface

6.40.1 CScalar class: instance creation

type Answer a CType for the receiver - for example, CByteType if the receiver is CByte.

value: anObject
Answer a newly allocated CObject containing the passed value, anObject. Remember to call #addToBeFinalized if you want the CObject to be automatically freed

6.40.2 CScalar: accessing

value Answer the value the receiver is pointing to. The exact returned value depends on the receiver's class

value: aValue
Set the receiver to point to the value, aValue. The exact meaning of aValue depends on the receiver's class

6.41 CScalarCType

Defined in namespace Smalltalk
Category: Language-C interface

6.41.1 CScalarCType: accessing

valueType valueType is used as a means to communicate to the interpreter the underlying type of the data. For scalars, it is supplied by the CObject subclass.

6.41.2 CScalarCType: storing

storeOn: aStream
Store Smalltalk code that compiles to the receiver

6.42 CShort

Defined in namespace Smalltalk
Category: Language-C interface

6.42.1 CShort class: accessing

alignof Answer the receiver's instances required aligment

scalarIndex
Private - Answer an index referring to the receiver's instances scalar type

sizeof Answer the receiver's instances size

6.42.2 CShort: accessing

alignof Answer the receiver's required aligment

scalarIndex
Private - Answer an index referring to the receiver's scalar type

sizeof Answer the receiver's size

6.43 CSmalltalk

Defined in namespace Smalltalk
Category: Language-C interface

6.43.1 CSmalltalk class: accessing

alignof Answer the receiver's instances required aligment

scalarIndex
Private - Answer an index referring to the receiver's instances scalar type

sizeof Answer the receiver's instances size

6.43.2 CSmalltalk: accessing

alignof Answer the receiver's required aligment

scalarIndex
Private - Answer an index referring to the receiver's scalar type

sizeof Answer the receiver's size

6.44 CString

Defined in namespace Smalltalk
Category: Language-C interface

Technically, CString is really a pointer to type char. However, it's so darn useful as a distinct datatype, and it is a separate datatype in Smalltalk, so we allow developers to express their semantics more precisely by using a more descriptive type.

In general, I behave like a cross between an array of characters and a pointer to a character. I provide the protocol for both data types. My #value method returns a Smalltalk String, as you would expect for a scalar datatype.

6.44.1 CString class: getting info

alignof Answer the receiver's instances required aligment

scalarIndex
Private - Answer an index referring to the receiver's instances scalar type

sizeof Answer the receiver's size

6.44.2 CString: accessing

alignof Answer the receiver's required aligment

scalarIndex
Private - Answer an index referring to the receiver's scalar type

sizeof Answer the receiver's size

6.44.3 CString: pointer like behavior

+ anInteger
Return another CString pointing at &receiver[anInteger] (or, if you prefer, what 'receiver + anInteger' does in C).

- intOrPtr If intOrPtr is an integer, return another CString pointing at &receiver[-anInteger] (or, if you prefer, what 'receiver - anInteger' does in C). If it is a CString, return the difference in chars, i.e. in bytes, between the two pointed addresses (or, if you prefer, what 'receiver - anotherCharPtr' does in C)

addressAt: anIndex
Access the string, returning a Smalltalk CChar corresponding to the given indexed element of the string. anIndex is zero-based, just like with all other C-style accessing.

at: anIndex
Access the string, returning the Smalltalk Character corresponding to the given indexed element of the string. anIndex is zero-based, just like with all other C-style accessing.

at: anIndex put: aCharacter
Store in the string a Smalltalk Character, at the given indexed element of the string. anIndex is zero-based, just like with all other C-style accessing.

decr Adjust the pointer by one byte down (i.e. –receiver)

decrBy: anInteger
Adjust the pointer by anInteger bytes down (i.e. receiver -= anInteger). Note that, unlike #-, #decrBy: does not support passing another CString as its parameter, since neither C supports something like 'charPtr -= anotherCharPtr'

deref Access the string, returning the Smalltalk CChar corresponding to the first element of the string. This may not make much sense, but it resembles what '*string' does in C.

deref: aCChar
Access the string, setting the first element of the string to the value of the passed CChar. This may not make much sense, but it resembles what we get in C if we do *string = 's'.

incr Adjust the pointer by one byte up (i.e. ++receiver)

incrBy: anInteger
Adjust the pointer by anInteger bytes up (i.e. receiver += anInteger)

replaceWith: aString
Overwrite memory starting at the receiver's address, with the contents of the Smalltalk String aString, null-terminating it. Ensure there is free space enough, or big trouble will hit you!

6.45 CStruct

Defined in namespace Smalltalk
Category: Language-C interface

6.45.1 CStruct class: subclass creation

compileDeclaration: array
Compile methods that implement the declaration in array.

6.46 CType

Defined in namespace Smalltalk
Category: Language-C interface

I am not part of the standard Smalltalk kernel class hierarchy. I contain type information used by subclasses of CObject, which represents external C data items.

My only instance variable, cObjectType, is used to hold onto the CObject subclass that gets created for a given CType. Used primarily in the C part of the interpreter because internally it cannot execute methods to get values, so it has a simple way to access instance variable which holds the desired subclass.

My subclasses have instances which represent the actual data types; for the scalar types, there is only one instance created of each, but for the aggregate types, there is at least one instance per base type and/or number of elements.

6.46.1 CType class: C instance creation

cObjectType: aCObjectSubclass
Create a new CType for the given subclass of CObject

6.46.2 CType: accessing

alignof Answer the size of the receiver's instances

arrayType: size
Answer a CArrayCType which represents an array with the given size of CObjects whose type is in turn represented by the receiver

cObjectType
Answer the CObject subclass whose instance is created when new is sent to the receiver

ptrType Answer a CPtrCType which represents a pointer to CObjects whose type is in turn represented by the receiver

sizeof Answer the size of the receiver's instances

valueType valueType is used as a means to communicate to the interpreter the underlying type of the data. For anything but scalars, it's just 'self'

6.46.3 CType: C instance creation

address: cObjOrInt
Create a new CObject with the type (class) identified by the receiver, pointing to the given address (identified by an Integer or CObject).

new Allocate a new CObject with the type (class) identified by the receiver. It is the caller's responsibility to free the memory allocated for it.

6.46.4 CType: storing

storeOn: aStream
Store Smalltalk code that compiles to the receiver

6.47 CUChar

Defined in namespace Smalltalk
Category: Language-C interface

6.47.1 CUChar class: getting info

alignof Answer the receiver's instances required aligment

scalarIndex
Private - Answer an index referring to the receiver's instances scalar type

sizeof Answer the receiver's instances size

6.47.2 CUChar: accessing

alignof Answer the receiver's required aligment

scalarIndex
Private - Answer an index referring to the receiver's scalar type

sizeof Answer the receiver's size

6.48 CUInt

Defined in namespace Smalltalk
Category: Language-C interface

6.48.1 CUInt class: accessing

alignof Answer the receiver's instances required aligment

scalarIndex
Private - Answer an index referring to the receiver's instances scalar type

sizeof Answer the receiver's instances size

6.48.2 CUInt: accessing

alignof Answer the receiver's required aligment

scalarIndex
Private - Answer an index referring to the receiver's scalar type

sizeof Answer the receiver's size

6.49 CULong

Defined in namespace Smalltalk
Category: Language-C interface

6.49.1 CULong class: accessing

alignof Answer the receiver's instances required aligment

scalarIndex
Private - Answer an index referring to the receiver's instances scalar type

sizeof Answer the receiver's instances size

6.49.2 CULong: accessing

alignof Answer the receiver's required aligment

scalarIndex
Private - Answer an index referring to the receiver's scalar type

sizeof Answer the receiver's size

6.50 CUnion

Defined in namespace Smalltalk
Category: Language-C interface

6.50.1 CUnion class: subclass creation

compileDeclaration: array
Compile methods that implement the declaration in array.

6.51 CUShort

Defined in namespace Smalltalk
Category: Language-C interface

6.51.1 CUShort class: accessing

alignof Answer the receiver's instances required aligment

scalarIndex
Private - Answer an index referring to the receiver's instances scalar type

sizeof Answer the receiver's instances size

6.51.2 CUShort: accessing

alignof Answer the receiver's required aligment

scalarIndex
Private - Answer an index referring to the receiver's scalar type

sizeof Answer the receiver's size

6.52 Date

Defined in namespace Smalltalk
Category: Language-Data types

My instances represent dates. My base date is defined to be Jan 1, 1901. I provide methods for instance creation (including via "symbolic" dates, such as "Date newDay: 14 month: #Feb year: 1990".

PLEASE BE WARNED – use this class only for dates after 1582 AD; that's the beginning of the epoch. Dates before 1582 will not be correctly printed. In addition, since ten days were lost from October 5 through October 15, operations between a Gregorian date (after 15-Oct-1582) and a Julian date (before 5-Oct-1582) will give incorrect results; or, 4-Oct-1582 + 2 days will yield 6-Oct-1582 (a non-existent day!), not 16-Oct-1582.

In fact, if you pass a year < 1582 to a method like #newDay:month:year: it will assume that it is a two-digit year (e.g. 90=1990, 1000=2900). The only way to create Julian calendar dates is with the #fromDays: instance creation method.

6.52.1 Date class: basic

abbreviationOfDay: dayIndex
Answer the abbreviated name of the day of week corresponding to the given index

dayOfWeek: dayName
Answer the index of the day of week corresponding to the given name

daysInMonth: monthName forYear: yearInteger
Answer the number of days in the given (named) month for the given year

daysInYear: i
Answer the number of days in the given year

indexOfMonth: monthName
Answer the index of the month corresponding to the given name

initDayNameDict
Initialize the DayNameDict to the names of the days

initialize Initialize the receiver

initMonthNameDict
Initialize the MonthNameDict to the names of the months

nameOfDay: dayIndex
Answer the name of the day of week corresponding to the given index

nameOfMonth: monthIndex
Answer the name of the month corresponding to the given index

shortNameOfMonth: monthIndex
Answer the name of the month corresponding to the given index

6.52.2 Date class: instance creation (ANSI)

year: y day: d hour: h minute: min second: s
Answer a Date denoting the d-th day of the given year

year: y month: m day: d hour: h minute: min second: s
Answer a Date denoting the d-th day of the given (as a number) month and year

6.52.3 Date class: instance creation (Blue Book)

dateAndTimeNow
Answer an array containing the current date and time

fromDays: dayCount
Answer a Date denoting dayCount days past 1/1/1901

fromJulian: jd
Answer a Date denoting the jd-th day in the astronomical Julian calendar.

fromSeconds: time
Answer a Date denoting the date time seconds past Jan 1st, 1901

newDay: day month: monthName year: yearInteger
Answer a Date denoting the dayCount day of the given (named) month and year

newDay: day monthIndex: monthIndex year: yearInteger
Answer a Date denoting the dayCount day of the given (as a number) month and year

newDay: dayCount year: yearInteger
Answer a Date denoting the dayCount day of the yearInteger year

readFrom: aStream
Parse an instance of the receiver from aStream

today Answer a Date denoting the current date in local time

utcDateAndTimeNow
Answer an array containing the current date and time in Coordinated Universal Time (UTC)

utcToday Answer a Date denoting the current date in Coordinated Universal Time (UTC)

6.52.4 Date: basic

addDays: dayCount
Answer a new Date pointing dayCount past the receiver

subtractDate: aDate
Answer the number of days between aDate and the receiver (negative if the receiver is before aDate)

subtractDays: dayCount
Answer a new Date pointing dayCount before the receiver

6.52.5 Date: compatibility (non-ANSI)

day Answer the day represented by the receiver

dayName Answer the day of week of the receiver as a Symbol

shortMonthName
Answer the abbreviated name of the month represented by the receiver

6.52.6 Date: date computations

asSeconds Answer the date as the number of seconds from 1/1/1901.

dayOfMonth
Answer the day represented by the receiver (same as #day)

dayOfWeek
Answer the day of week of the receiver. 1 = Monday, 7 = Sunday

dayOfWeekAbbreviation
Answer the day of week of the receiver as a Symbol

dayOfWeekName
Answer the day of week of the receiver as a Symbol

dayOfYear
Answer the days passed since 31/12 of last year; e.g. New Year's Day is 1

daysFromBaseDay
Answer the days passed since 1/1/1901

daysInMonth
Answer the days in the month represented by the receiver

daysInYear
Answer the days in the year represented by the receiver

daysLeftInMonth
Answer the days to the end of the month represented by the receiver

daysLeftInYear
Answer the days to the end of the year represented by the receiver

firstDayOfMonth
Answer a Date representing the first day of the month represented by the receiver

isLeapYear
Answer whether the receiver refers to a date in a leap year.

lastDayOfMonth
Answer a Date representing the last day of the month represented by the receiver

month Answer the month represented by the receiver

monthAbbreviation
Answer the abbreviated name of the month represented by the receiver

monthName
Answer the name of the month represented by the receiver

year Answer the year represented by the receiver

6.52.7 Date: printing

printOn: aStream
Print a representation for the receiver on aStream

6.52.8 Date: storing

storeOn: aStream
Store on aStream Smalltalk code compiling to the receiver

6.52.9 Date: testing

< aDate Answer whether the receiver indicates a date preceding aDate

= aDate Answer whether the receiver indicates the same date as aDate

hash Answer an hash value for the receievr

6.53 DateTime

Defined in namespace Smalltalk
Category: Language-Data types
My instances represent timestamps.

6.53.1 DateTime class: information

clockPrecision
Answer 'ClockPrecision'.

initialize Initialize the receiver's class variables

6.53.2 DateTime class: instance creation

now Answer an instance of the receiver referring to the current date and time.

readFrom: aStream
Parse an instance of the receiver from aStream

year: y day: d hour: h minute: min second: s
Answer a DateTime denoting the d-th day of the given year, and setting the time part to the given hour, minute, and second

year: y day: d hour: h minute: min second: s offset: ofs
Answer a DateTime denoting the d-th day of the given year. Set the offset field to ofs (a Duration), and the time part to the given hour, minute, and second

year: y month: m day: d hour: h minute: min second: s
Answer a DateTime denoting the d-th day of the given (as a number) month and year, setting the time part to the given hour, minute, and second

year: y month: m day: d hour: h minute: min second: s offset: ofs
Answer a DateTime denoting the d-th day of the given (as a number) month and year. Set the offset field to ofs (a Duration), and the the time part to the given hour, minute, and second

6.53.3 DateTime class: instance creation (non-ANSI)

fromDays: days seconds: secs offset: ofs
Answer a DateTime denoting the d-th day of the given (as a number) month and year. Set the offset field to ofs (a Duration), and the the time part to the given hour, minute, and second

6.53.4 DateTime: basic

+ aDuration
Answer a new Date pointing dayCount past the receiver

- aDateTimeOrDuration
Answer a new Date pointing dayCount before the receiver

6.53.5 DateTime: computations

asSeconds Answer the date as the number of seconds from 1/1/1901.

dayOfWeek
Answer the day of week of the receiver. Unlike Dates, DateAndTimes have 1 = Sunday, 7 = Saturday

hour Answer the hour in a 24-hour clock

hour12 Answer the hour in a 12-hour clock

hour24 Answer the hour in a 24-hour clock

meridianAbbreviation
Answer either #AM (for anti-meridian) or #PM (for post-meridian)

minute Answer the minute

second Answer the month represented by the receiver

6.53.6 DateTime: printing

printOn: aStream
Print a representation for the receiver on aStream

6.53.7 DateTime: splitting in dates & times

asDate Answer a Date referring to the same day as the receiver

asTime Answer a Time referring to the same time (from midnight) as the receiver

at: anIndex
Since in the past timestamps were referred to as Arrays containing a Date and a Time (in this order), this method provides access to DateTime objects like if they were two-element Arrays.

6.53.8 DateTime: storing

storeOn: aStream
Store on aStream Smalltalk code compiling to the receiver

6.53.9 DateTime: testing

< aDateTime
Answer whether the receiver indicates a date preceding aDate

= aDateTime
Answer whether the receiver indicates the same date as aDate

hash Answer an hash value for the receievr

6.53.10 DateTime: time zones

asLocal Answer the receiver, since DateTime objects store themselves in Local time

asUTC Convert the receiver to UTC time, and answer a new DateTime object.

offset Answer the receiver's offset from UTC to local time (e.g. +3600 seconds for Central Europe Time, -3600*6 seconds for Eastern Standard Time). The offset is expressed as a Duration

offset: anOffset
Answer a copy of the receiver with the offset from UTC to local time changed to anOffset (a Duration).

timeZoneAbbreviation
Answer an abbreviated indication of the receiver's offset, expressed as 'shhmm', where 'hh' is the number of hours and 'mm' is the number of minutes between UTC and local time, and 's' can be '+' for the Eastern hemisphere and '-' for the Western hemisphere.

timeZoneName
Answer the time zone name for the receiver (currently, it is simply 'GMT +xxxx', where 'xxxx' is the receiver's #timeZoneAbbreviation).

6.54 Delay

Defined in namespace Smalltalk
Category: Language-Processes
I am the ultimate agent for frustration in the world. I cause things to wait (typically much more than is appropriate, but it is those losing operating systems' fault). When a process sends one of my instances a wait message, that process goes to sleep for the interval specified when the instance was created.

6.54.1 Delay class: general inquiries

millisecondClockValue
Private - Answer the number of milliseconds since midnight

6.54.2 Delay class: initialization

initialize Private - Initialize the receiver and the associated process

6.54.3 Delay class: instance creation

forMilliseconds: millisecondCount
Answer a Delay waiting for millisecondCount milliseconds

forSeconds: secondCount
Answer a Delay waiting for secondCount seconds

untilMilliseconds: millisecondCount
Answer a Delay waiting for millisecondCount milliseconds after midnight

6.54.4 Delay: accessing

resumptionTime
Answer the time when a process waiting on a Delay will resume

6.54.5 Delay: comparing

= aDelay Answer whether the receiver and aDelay denote the same delay

hash Answer an hash value for the receiver

6.54.6 Delay: process delay

wait Wait until the amount of time represented by the instance of Delay elapses

6.55 DelayedAdaptor

Defined in namespace Smalltalk
Category: Language-Data types

I can be used where many expensive updates must be performed. My in- stances buffer the last value that was set, and only actually set the value when the #trigger message is sent. Apart from this, I'm equi- valent to PluggableAdaptor.

6.55.1 DelayedAdaptor: accessing

trigger Really set the value of the receiver.

value Get the value of the receiver.

value: anObject

Set the value of the receiver - actually, the value is cached and is not set until the #trigger method is sent.

6.56 Dictionary

Defined in namespace Smalltalk
Category: Collections-Keyed

I implement a dictionary, which is an object that is indexed by unique objects (typcially instances of Symbol), and associates another object with that index. I use the equality operator = to determine equality of indices.

6.56.1 Dictionary class: instance creation

new Create a new dictionary with a default size

6.56.2 Dictionary: accessing

add: newObject
Add the newObject association to the receiver

associationAt: key
Answer the key/value Association for the given key. Fail if the key is not found

associationAt: key ifAbsent: aBlock
Answer the key/value Association for the given key. Evaluate aBlock (answering the result) if the key is not found

at: key Answer the value associated to the given key. Fail if the key is not found

at: key ifAbsent: aBlock
Answer the value associated to the given key, or the result of evaluating aBlock if the key is not found

at: aKey ifAbsentPut: aBlock
Answer the value associated to the given key. If the key is not found, evaluate aBlock and associate the result to aKey before returning.

at: aKey ifPresent: aBlock
If aKey is absent, answer nil. Else, evaluate aBlock passing the associated value and answer the result of the invocation

at: key put: value
Store value as associated to the given key

keyAtValue: value
Answer the key associated to the given value. Evaluate exceptionBlock (answering the result) if the value is not found

keyAtValue: value ifAbsent: exceptionBlock
Answer the key associated to the given value. Evaluate exceptionBlock (answering the result) if the value is not found. IMPORTANT: == is used to compare values

keys Answer a kind of Set containing the keys of the receiver

values Answer a Bag containing the values of the receiver

6.56.3 Dictionary: awful ST-80 compatibility hacks

findKeyIndex: key
Tries to see if key exists as a the key of an indexed variable. As soon as nil or an association with the correct key is found, the index of that slot is answered

6.56.4 Dictionary: dictionary enumerating

associationsDo: aBlock
Pass each association in the dictionary to aBlock

collect: aBlock
Answer a new dictionary where the keys are the same and the values are obtained by passing each value to aBlock and collecting the return values

do: aBlock
Pass each value in the dictionary to aBlock

keysAndValuesDo: aBlock
Pass each key/value pair in the dictionary as two distinct parameters to aBlock

keysDo: aBlock
Pass each key in the dictionary to aBlock

reject: aBlock
Answer a new dictionary containing the key/value pairs for which aBlock returns false. aBlock only receives the value part of the pairs.

select: aBlock
Answer a new dictionary containing the key/value pairs for which aBlock returns true. aBlock only receives the value part of the pairs.

6.56.5 Dictionary: dictionary removing

remove: anObject
This method should not be called for instances of this class.

remove: anObject ifAbsent: aBlock
This method should not be called for instances of this class.

removeAllKeys: keys
Remove all the keys in keys, without raising any errors

removeAllKeys: keys ifAbsent: aBlock
Remove all the keys in keys, passing the missing keys as parameters to aBlock as they're encountered

removeAssociation: anAssociation
Remove anAssociation's key from the dictionary

removeKey: key
Remove the passed key from the dictionary, fail if it is not found

removeKey: key ifAbsent: aBlock
Remove the passed key from the dictionary, answer the result of evaluating aBlock if it is not found

6.56.6 Dictionary: dictionary testing

includes: anObject
Answer whether the receiver contains anObject as one of its values

includesAssociation: anAssociation
Answer whether the receiver contains the key which is anAssociation's key and its value is anAssociation's value

includesKey: key
Answer whether the receiver contains the given key

occurrencesOf: aValue
Answer whether the number of occurrences of aValue as one of the receiver's values

6.56.7 Dictionary: polymorphism hacks

withAllSuperspaces
This method is needed by the compiler

6.56.8 Dictionary: printing

printOn: aStream
Print a representation of the receiver on aStream

6.56.9 Dictionary: storing

storeOn: aStream
Print Smalltalk code compiling to the receiver on aStream

6.56.10 Dictionary: testing

= aDictionary
Answer whether the receiver and aDictionary are equal

hash Answer the hash value for the receiver

6.57 DirectedMessage

Defined in namespace Smalltalk
Category: Language-Implementation
I represent a message send: I contain the receiver, selector and arguments for a message.

6.57.1 DirectedMessage class: creating instances

selector: aSymbol arguments: anArray
This method should not be called for instances of this class.

selector: aSymbol arguments: anArray receiver: anObject
Create a new instance of the receiver

6.57.2 DirectedMessage: accessing

receiver Answer the receiver

receiver: anObject
Change the receiver

6.57.3 DirectedMessage: basic

printOn: aStream
Print a representation of the receiver on aStream

send Send the message

6.57.4 DirectedMessage: saving and loading

reconstructOriginalObject
This method is used when DirectedMessages are used together with PluggableProxies (see ObjectDumper). It sends the receiver to reconstruct the object that was originally stored.

6.58 Directory

Defined in namespace Smalltalk
Category: Streams-Files

6.58.1 Directory class: C functions

primCreate: dirName mode: mode
C call-out to mkdir. Do not modify!

primRemove: fileName
C call-out to rmdir. Do not modify!

primWorking: dirName
C call-out to chdir. Do not modify!

working C call-out to getCurDirName. Do not modify!

6.58.2 Directory class: file name management

append: fileName to: directory
Answer the name of a file named 'fileName' which resides in a directory named 'directory'.

pathSeparator
Answer (as a Character) the character used to separate directory names

pathSeparatorString
Answer (in a String) the character used to separate directory names

6.58.3 Directory class: file operations

create: dirName
Change the current working directory to dirName.

working: dirName
Change the current working directory to dirName.

6.58.4 Directory class: reading system defaults

home Answer the path to the user's home directory

image Answer the path to GNU Smalltalk's image file

kernel Answer the path in which a local version of the GNU Smalltalk kernel's Smalltalk source files were searched when the image was created

localKernel
Answer the path in which a local version of the GNU Smalltalk kernel's Smalltalk source files were found

module Answer the path to GNU Smalltalk's dynamically loaded modules

systemKernel
Answer the path to the GNU Smalltalk kernel's Smalltalk source files

6.58.5 Directory: accessing

at: aName
Answer a File object for a file named 'aName' residing in the directory represented by the receiver.

directoryAt: aName
Answer a Directory object for a file named 'aName' residing in the directory represented by the receiver.

fullNameAt: aName
Answer a String containing the full path to a file named 'aName' which resides in the directory represented by the receiver.

includes: aName
Answer whether a file named 'aName' exists in the directory represented by the receiver.

nameAt: aName
Answer a String containing the path to a file named 'aName' which resides in the directory represented by the receiver.

6.58.6 Directory: C functions

extractDirentName: dirent
C call-out to extractDirentName. Do not modify!

readDir: dirObject
C call-out to readdir. Do not modify!

rewindDir: dirObject
C call-out to rewinddir. Do not modify!

6.58.7 Directory: enumerating

contents Answer an Array with the names of the files in the directory represented by the receiver.

do: aBlock
Evaluate aBlock once for each file in the directory represented by the receiver, passing its name. aBlock should not return.

filesMatching: aPattern do: block
Evaluate block on the File objects that match aPattern (according to String>>-#match:) in the directory named by the receiver.

namesMatching: aPattern do: block
Evaluate block on the file names that match aPattern (according to String>>-#match:) in the directory named by the receiver.

6.59 DLD

Defined in namespace Smalltalk
Category: Language-C interface
...and Gandalf said: "Many folk like to know beforehand what is to be set on the table; but those who have laboured to prepare the feast like to keep their secret; for wonder makes the words of praise louder."

I am just an ancillary class used to reference some C functions. Most of my actual functionality is used by redefinitions of methods in CFunctionDescriptor and Behavior.

6.59.1 DLD class: C functions

defineCFunc: aName as: aFuncAddr
C call-out to defineCFunc. Do not modify!

library: libHandle getFunc: aFuncString
C call-out to dldGetFunc. Do not modify!

linkFile: aFileName
C call-out to dldLink. Do not modify!

6.59.2 DLD class: Dynamic Linking

addLibrary: library
Add library to the search path of libraries to be used by DLD.

addModule: library
Add library to the list of modules to be loaded when the image is started. The gst_initModule function in the library is called, but the library will not be put in the search path used whenever a C function is requested but not registered.

defineExternFunc: aFuncName
This method calls #primDefineExternFunc: to try to link to a function with the given name, and answers whether the linkage was successful. You can redefine this method to restrict the ability to do dynamic linking.

initialize Private - Initialize the receiver's class variables

libraryList
Answer a copy of the search path of libraries to be used by DLD

moduleList
Answer a copy of the modules reloaded when the image is started

primDefineExternFunc: aFuncName
This method tries to link to a function with the given name, and answers whether the linkage was successful. It should not be overridden.

update: aspect
Called on startup - Make DLD re-link and reset the addresses of all the externally defined functions

6.60 DumperProxy

Defined in namespace Smalltalk
Category: Streams-Files
I am an helper class for ObjectDumper. When an object cannot be saved in the standard way, you can register a subclass of me to provide special means to save that object.

6.60.1 DumperProxy class: accessing

acceptUsageForClass: aClass
The receiver was asked to be used as a proxy for the class aClass. Answer whether the registration is fine. By default, answer true

loadFrom: anObjectDumper
Reload a proxy stored in anObjectDumper and reconstruct the object

6.60.2 DumperProxy class: instance creation

on: anObject
Answer a proxy to be used to save anObject. This method MUST be overridden and anObject must NOT be stored in the object's instance variables unless you override #dumpTo:, because that would result in an infinite loop!

6.60.3 DumperProxy: saving and restoring

dumpTo: anObjectDumper
Dump the proxy to anObjectDumper – the #loadFrom: class method will reconstruct the original object.

object Reconstruct the object stored in the proxy and answer it

6.61 Duration

Defined in namespace Smalltalk
Category: Language-Data types

6.61.1 Duration class: instance creation

days: d Answer a duration of 'd' days

days: d hours: h minutes: m seconds: s
Answer a duration of 'd' days and the given number of hours, minutes, and seconds.

initialize Initialize the receiver's instance variables

zero Answer a duration of zero seconds.

6.61.2 Duration class: instance creation (non ANSI)

fromDays: days seconds: secs offset: unused
Answer a duration of 'd' days and 'secs' seconds. The last parameter is unused; this message is available for interoperability with the DateTime class.

6.61.3 Duration: arithmetics

*** factor** Answer a Duration that is 'factor' times longer than the receiver

+ aDuration
Answer a Duration that is the sum of the receiver and aDuration's lengths.

- aDuration
Answer a Duration that is the difference of the receiver and aDuration's lengths.

/ factorOrDuration
: If the parameter is a Duration, answer the ratio between the receiver and factorOrDuration. Else divide the receiver by factorOrDuration (a Number) and answer a new Duration that is correspondingly shorter.

abs Answer a Duration that is as long as the receiver, but always in the future.

days Answer the number of days in the receiver

negated Answer a Duration that is as long as the receiver, but with past and future exchanged.

negative Answer whether the receiver is in the past.

positive Answer whether the receiver is a zero-second duration or is in the future.

printOn: aStream
: Print a represention of the receiver on aStream.

6.62 Error

Defined in namespace Smalltalk
Category: Language-Exceptions
: Error represents a fatal error. Instances of it are not resumable.

6.62.1 Error: exception description

description
: Answer a textual description of the exception.

isResumable
: Answer false. Error exceptions are by default unresumable; subclasses can override this method if desired.

6.63 Exception

Defined in namespace Smalltalk
Category: Language-Exceptions
: An Exception defines the characteristics of an exceptional event in a different way than CoreExceptions. Instead of creating an hierarchy of objects and setting attributes of the objects, you create an hierarchy of classes and override methods in those classes; instances of those classes are passed to the handlers instead of instances of the common class Signal.

  Internally, Exception and every subclass of it hold onto a CoreException, so the two mechanisms are actually interchangeable.

6.63.1 Exception class: comparison

handles: anException
: Answer whether the receiver handles ‘anException’.

6.63.2 Exception class: creating ExceptionCollections

, aTrappableEvent
Answer an ExceptionCollection containing all the exceptions in the receiver and all the exceptions in aTrappableEvent

6.63.3 Exception class: initialization

initialize Initialize the 'links' between the core exception handling system and the ANSI exception handling system.

6.63.4 Exception class: instance creation

new Create an instance of the receiver, which you will be able to signal later.

signal Create an instance of the receiver, give it default attributes, and signal it immediately.

signal: messageText
Create an instance of the receiver, set its message text, and signal it immediately.

6.63.5 Exception class: interoperability with TrappableEvents

allExceptionsDo: aBlock
Private - Pass the coreException to aBlock

coreException
Private - Answer the coreException which represents instances of the receiver

whenSignalledIn: onDoBlock do: handlerBlock exitBlock: exitBlock
Private - Create an ExceptionHandler from the arguments and register it

6.63.6 Exception: comparison

= anObject
Answer whether the receiver is equal to anObject. This is true if either the receiver or its coreException are the same object as anObject.

hash Answer an hash value for the receiver.

6.63.7 Exception: exception description

defaultAction
Execute the default action that is attached to the receiver.

description
Answer a textual description of the exception.

isResumable
Answer true. Exceptions are by default resumable.

6.63.8 Exception: exception signaling

signal Raise the exceptional event represented by the receiver

signal: messageText
Raise the exceptional event represented by the receiver, setting its message text to messageText.

6.64 ExceptionSet

Defined in namespace Smalltalk
Category: Language-Exceptions
My instances are not real exceptions: they can only be used as arguments to #on:do:... methods in BlockClosure. They act as shortcuts that allows you to use the same handler for many exceptions without having to write duplicate code

6.64.1 ExceptionSet class: instance creation

new Private - Answer a new, empty ExceptionSet

6.64.2 ExceptionSet: enumerating

allExceptionsDo: aBlock
Private - Evaluate aBlock for every exception in the receiver. Answer the receiver

handles: exception
Answer whether the receiver handles 'exception'.

6.65 False

Defined in namespace Smalltalk
Category: Language-Data types
I always tell lies. I have a single instance in the system, which represents the value false.

6.65.1 False: basic

& aBoolean
We are false – anded with anything, we always answer false

and: aBlock
We are false – anded with anything, we always answer false

eqv: aBoolean
Answer whether the receiver and aBoolean represent the same boolean value

ifFalse: falseBlock
We are false – evaluate the falseBlock

ifFalse: falseBlock ifTrue: trueBlock
We are false – evaluate the falseBlock

ifTrue: trueBlock
We are false – answer nil

ifTrue: trueBlock ifFalse: falseBlock
We are false – evaluate the falseBlock

not We are false – answer true

or: aBlock We are false – ored with anything, we always answer the other operand, so evaluate aBlock

xor: aBoolean
Answer whether the receiver and aBoolean represent different boolean values

| aBoolean
We are false – ored with anything, we always answer the other operand

6.65.2 False: C hacks

asCBooleanValue
Answer '0'.

6.65.3 False: printing

printOn: aStream
Print a representation of the receiver on aStream

6.66 File

Defined in namespace Smalltalk
Category: Streams-Files

6.66.1 File class: C functions

errno C call-out to errno. Do not modify!

primRemove: fileName
C call-out to unlink. Do not modify!

primRename: oldFileName to: newFileName
C call-out to rename. Do not modify!

stringError: errno
C call-out to strerror. Do not modify!

6.66.2 File class: file name management

extensionFor: aString
Answer the extension of a file named 'aString'

fullNameFor: aString
Answer the full path to a file called 'aString', resolving the '.' and '..' directory entries, and answer the result. Answer nil if the file is invalid (such as '/usr/../..')

pathFor: aString
Answer the path of the name of a file called 'aString', and answer the result

stripExtensionFrom: aString
Remove the extension from the name of a file called 'aString', and answer the result

stripPathFrom: aString
Remove the path from the name of a file called 'aString', and answer the file name plus extension.

6.66.3 File class: file operations

checkError
Return whether an error had been reported or not. If there had been one, raise an exception too

checkError: errno
The error with the C code 'errno' has been reported. If errno >= 1, raise an exception

remove: fileName
Remove the file with the given path name

rename: oldFileName to: newFileName
Rename the file with the given path name oldFileName to newFileName

6.66.4 File class: initialization

initialize Initialize the receiver's class variables

6.66.5 File class: instance creation

name: aName
Answer a new file with the given path. The path is not validated until some of the fields of the newly created objects are accessed

6.66.6 File class: reading system defaults

image Answer the full path to the image being used.

6.66.7 File class: testing

exists: fileName
Answer whether a file with the given name exists

isAccessible: fileName
Answer whether a directory with the given name exists and can be accessed

isExecutable: fileName
Answer whether a file with the given name exists and can be executed

isReadable: fileName
Answer whether a file with the given name exists and is readable

isWriteable: fileName
Answer whether a file with the given name exists and is writeable

6.66.8 File: accessing

creationTime
Answer the creation time of the file identified by the receiver. On some operating systems, this could actually be the last change time (the 'last change time' has to do with permissions, ownership and the like).

lastAccessTime
Answer the last access time of the file identified by the receiver

lastChangeTime
Answer the last change time of the file identified by the receiver (the 'last change time' has to do with permissions, ownership and the like). On some operating systems, this could actually be the file creation time.

lastModifyTime
Answer the last modify time of the file identified by the receiver (the 'last modify time' has to do with the actual file contents).

name Answer the name of the file identified by the receiver

refresh Refresh the statistics for the receiver

size Answer the size of the file identified by the receiver

6.66.9 File: C functions

closeDir: dirObject
C call-out to closedir. Do not modify!

openDir: dirName
C call-out to opendir. Do not modify!

primIsExecutable: name
C call-out to fileIsExecutable. Do not modify!

primIsReadable: name
C call-out to fileIsReadable. Do not modify!

primIsWriteable: name
C call-out to fileIsWriteable. Do not modify!

statOn: fileName into: statStruct
C call-out to stat. Do not modify!

6.66.10 File: file name management

extension Answer the extension of the receiver

fullName Answer the full name of the receiver, resolving the '.' and '..' directory entries, and answer the result. Answer nil if the name is invalid (such as '/usr/../../badname')

path Answer the path (if any) of the receiver

stripExtension
Answer the path (if any) and file name of the receiver

stripPath Answer the file name and extension (if any) of the receiver

6.66.11 File: file operations

contents Open a read-only FileStream on the receiver, read its contents, close the stream and answer the contents

open: mode
Open the receiver in the given mode (as answered by FileStream's class constant methods)

readStream
Open a read-only FileStream on the receiver

remove Remove the file identified by the receiver

renameTo: newName
Remove the file identified by the receiver

writeStream
Open a write-only FileStream on the receiver

6.66.12 File: testing

exists Answer whether a file with the name contained in the receiver does exist.

isAccessible
Answer whether a directory with the name contained in the receiver does exist and can be accessed

isDirectory
Answer whether a file with the name contained in the receiver does exist and identifies a directory.

isExecutable
Answer whether a file with the name contained in the receiver does exist and is executable

isFile Answer whether a file with the name contained in the receiver does exist and does not identify a directory.

isReadable
Answer whether a file with the name contained in the receiver does exist and is readable

isWriteable
Answer whether a file with the name contained in the receiver does exist and is writeable

6.67 FileDescriptor

Defined in namespace Smalltalk
Category: Streams-Files
My instances are what conventional programmers think of as files. My instance creation methods accept the name of a disk file (or any named file object, such as /dev/rmt0 on UNIX or MTA0: on VMS).

6.67.1 FileDescriptor class: initialization

initialize Initialize the receiver's class variables

update: aspect
Close open files before quitting

6.67.2 FileDescriptor class: instance creation

append Open for writing. The file is created if it does not exist. The stream is positioned at the end of the file.

create Open for reading and writing. The file is created if it does not exist, otherwise it is truncated. The stream is positioned at the beginning of the file.

on: fd Open a FileDescriptor on the given file descriptor. Read-write access is assumed.

open: fileName
Open fileName in read-write mode - fail if the file cannot be opened. Else answer a new FileStream. The file will be automatically closed upon GC if the object is not referenced anymore, but you should close it with #close anyway. To keep a file open, send it #removeToBeFinalized

open: fileName mode: fileMode
Open fileName in the required mode - answered by #append, #create, #readWrite, #read or #write - and fail if the file cannot be opened. Else answer a new FileStream. For mode anyway you can use any standard C non-binary fopen mode. The file will be automatically closed upon GC if the object is not referenced anymore, but it is better to close it as soon as you're finished with it anyway, using #close. To keep a file open even when no references exist anymore, send it #removeToBeFinalized

open: fileName mode: fileMode ifFail: aBlock
Open fileName in the required mode - answered by #append, #create, #readWrite, #read or #write - and evaluate aBlock if the file cannot be opened. Else answer a new FileStream. For mode anyway you can use any The file will be automatically closed upon GC if the object is not referenced anymore, but it is better to close it as soon as you're finished with it anyway, using #close. To keep a file open even when no references exist anymore, send it #removeToBeFinalized

popen: commandName dir: direction
Open a pipe on the given command and fail if the file cannot be opened. Else answer a new FileStream. The pipe will not be automatically closed upon GC, even if the object is not referenced anymore, because when you close a pipe you have to wait for the associated process to terminate. To enforce automatic closing of the pipe, send it #addToBeFinalized. direction is returned by #read or #write ('r' or 'w') and is interpreted from the point of view of Smalltalk: reading means Smalltalk reads the standard output of the command, writing means Smalltalk writes the standard input of the command. The other channel (stdin when reading, stdout when writing) is the same as GST's, unless commandName alters it.

popen: commandName dir: direction ifFail: aBlock
Open a pipe on the given command and evaluate aBlock file cannot be opened. Else answer a new FileStream. The pipe will not be automatically closed upon GC, even if the object is not referenced anymore, because when you close a pipe you have to wait for the associated process to terminate. To enforce automatic closing of the pipe, send it #addToBeFinalized. direction is interpreted from the point of view of Smalltalk: reading means that Smalltalk reads the standard output of the command, writing means that Smalltalk writes the standard input of the command

read Open text file for reading. The stream is positioned at the beginning of the file.

readWrite Open for reading and writing. The stream is positioned at the beginning of the file.

write Truncate file to zero length or create text file for writing. The stream is positioned at the beginning of the file.

6.67.3 FileDescriptor: accessing

canRead Answer whether the file is open and we can read from it

canWrite Answer whether the file is open and we can write from it

ensureReadable
If the file is open, wait until data can be read from it. The wait allows other Processes to run.

ensureWriteable
If the file is open, wait until we can write to it. The wait allows other Processes to run.

exceptionalCondition
Answer whether the file is open and an exceptional condition (such as presence of out of band data) has occurred on it

fd Return the OS file descriptor of the file

isOpen Answer whether the file is still open

isPipe Answer whether the file is a pipe or an actual disk file

name Return the name of the file

waitForException
If the file is open, wait until an exceptional condition (such as presence of out of band data) has occurred on it. The wait allows other Processes to run.

6.67.4 FileDescriptor: basic

close Close the file

contents Answer the whole contents of the file

copyFrom: from to: to
Answer the contents of the file between the two given positions

finalize Close the file if it is still open by the time the object becomes garbage.

invalidate Invalidate a file descriptor

next Return the next character in the file, or nil at eof

nextByte Return the next byte in the file, or nil at eof

nextPut: aCharacter
Store aCharacter on the file

nextPutByte: anInteger
Store the byte, anInteger, on the file

nextPutByteArray: aByteArray
Store aByteArray on the file

position Answer the zero-based position from the start of the file

position: n
Set the file pointer to the zero-based position n

reset Reset the stream to its beginning

size Return the current size of the file, in bytes

truncate Truncate the file at the current position

6.67.5 FileDescriptor: built ins

fileOp: ioFuncIndex
Private - Used to limit the number of primitives used by FileStreams

fileOp: ioFuncIndex ifFail: aBlock
Private - Used to limit the number of primitives used by FileStreams.

fileOp: ioFuncIndex with: arg1
Private - Used to limit the number of primitives used by FileStreams

fileOp: ioFuncIndex with: arg1 ifFail: aBlock
Private - Used to limit the number of primitives used by FileStreams.

fileOp: ioFuncIndex with: arg1 with: arg2
Private - Used to limit the number of primitives used by FileStreams

fileOp: ioFuncIndex with: arg1 with: arg2 ifFail: aBlock
Private - Used to limit the number of primitives used by FileStreams.

fileOp: ioFuncIndex with: arg1 with: arg2 with: arg3
Private - Used to limit the number of primitives used by FileStreams

fileOp: ioFuncIndex with: arg1 with: arg2 with: arg3 ifFail: aBlock
Private - Used to limit the number of primitives used by FileStreams.

6.67.6 FileDescriptor: class type methods

isBinary We answer characters, so answer false

isExternalStream
We stream on an external entity (a file), so answer true

isText We answer characters, so answer true

6.67.7 FileDescriptor: initialize-release

initialize Initialize the receiver's instance variables

newBuffer Private - Answer a String to be used as the receiver's buffer

nextHunk Answer the next buffers worth of stuff in the Stream represented by the receiver. Do at most one actual input operation.

6.67.8 FileDescriptor: low-level access

read: byteArray
Ignoring any buffering, try to fill byteArray with the contents of the file

read: byteArray from: position to: end
Ignoring any buffering, try to fill the given range of byteArray with the contents of the file

read: byteArray numBytes: anInteger
Ignoring any buffering, try to fill anInteger bytes of byteArray with the contents of the file

write: byteArray
Ignoring any buffering, try to write the contents of byteArray in the file

write: byteArray from: position to: end
Ignoring any buffering, try to write to the file the given range of byteArray, starting at index position.

write: byteArray numBytes: anInteger
Ignoring any buffering, try to write to the file the first anInteger bytes of byteArray

6.67.9 FileDescriptor: overriding inherited methods

isEmpty Answer whether the receiver is empty

next: anInteger
Return the next 'anInteger' characters from the stream, as a String.

nextByteArray: anInteger
Return the next 'anInteger' bytes from the stream, as a ByteArray.

nextPutAll: aCollection
Put all the characters in aCollection in the file

reverseContents
Return the contents of the file from the last byte to the first

setToEnd Reset the file pointer to the end of the file

skip: anInteger
Skip anInteger bytes in the file

6.67.10 FileDescriptor: printing

printOn: aStream
Print a representation of the receiver on aStream

6.67.11 FileDescriptor: testing

atEnd Answer whether data has come to an end

6.68 FileSegment

Defined in namespace Smalltalk
Category: Language-Implementation

My instances represent sections of files. I am primarily used by the compiler to record source code locations. I am not a part of the normal Smalltalk-80 kernel; I am specific to the GNU Smalltalk implementation.

6.68.1 FileSegment class: basic

on: aFile startingAt: startPos for: sizeInteger

Create a new FileSegment referring to the contents of the given file, from the startPos-th byte and for sizeInteger bytes

6.68.2 FileSegment: basic

asString Answer a String containing the required segment of the file

fileName Answer the name of the file containing the segment

filePos Answer the position in the file where the segment starts

size Answer the length of the segment

withFileDo: aBlock

Evaluate aBlock passing it the FileStream in which the segment identified by the receiver is stored

6.68.3 FileSegment: equality

= aFileSegment

Answer whether the receiver and aFileSegment are equal.

hash Answer an hash value for the receiver.

6.69 FileStream

Defined in namespace Smalltalk
Category: Streams-Files

My instances are what conventional programmers think of as files. My instance creation methods accept the name of a disk file (or any named file object, such as /dev/rmt0 on UNIX or MTA0: on VMS).

6.69.1 FileStream class: file-in

fileIn: aFileName
File in the aFileName file. During a file in operation, global variables (starting with an uppercase letter) that are not declared yet don't yield an 'unknown variable' error. Instead, they are defined as nil in the 'Undeclared' dictionary (a global variable residing in Smalltalk). As soon as you add the variable to a namespace (for example by creating a class) the Association will be removed from Undeclared and reused in the namespace, so that the old references will automagically point to the new value.

fileIn: aFileName ifMissing: aSymbol
Conditionally do a file in, only if the key (often a class) specified by 'aSymbol' is not present in the Smalltalk system dictionary already. During a file in operation, global variables (starting with an uppercase letter) that are not declared don't yield an 'unknown variable' error. Instead, they are defined as nil in the 'Undeclared' dictionary (a global variable residing in Smalltalk). As soon as you add the variable to a namespace (for example by creating a class) the Association will be removed from Undeclared and reused in the namespace, so that the old references will automagically point to the new value.

fileIn: aFileName ifTrue: aBoolean
Conditionally do a file in, only if the supplied boolean is true. During a file in operation, global variables (starting with an uppercase letter) that are not declared don't yield an 'unknown variable' error. Instead, they are defined as nil in the 'Undeclared' dictionary (a global variable residing in Smalltalk). As soon as you add the variable to a namespace (for example by creating a class) the Association will be removed from Undeclared and reused in the namespace, so that the old references will automagically point to the new value.

fileIn: aFileName line: lineInteger from: realFileName at: aCharPos
File in the aFileName file giving errors such as if it was loaded from the given line, file name and starting position (instead of 1).

generateMakefileOnto: aStream
Generate a make file for the file-ins since record was last set to true. Store it on aStream

initialize Private - Initialize the receiver's class variables

record: recordFlag
Set whether Smalltalk should record information aboutnested file-ins. When recording is enabled, use #generateMakefileOnto: to automatically generate a valid makefile for the intervening file-ins.

require: assoc
Conditionally do a file in from the value of assoc, only if the key of assoc is not present in the Smalltalk system dictionary already. During a file in operation, global variables (starting with an uppercase letter) that are not declared don't yield an 'unknown variable' error. Instead, they are defined as

nil in the 'Undeclared' dictionary (a global variable residing in Smalltalk). As soon as you add the variable to a namespace (for example by creating a class) the Association will be removed from Undeclared and reused in the namespace, so that the old references will automagically point to the new value.

verbose: verboseFlag
Set whether Smalltalk should output debugging messages when filing in

6.69.2 FileStream class: standard streams

stderr Answer a FileStream that is attached the Smalltalk program's standard error file handle, which can be used for error messages and diagnostics issued by the program.

stdin Answer a FileStream that is attached the Smalltalk program's standard input file handle, which is the normal source of input for the program.

stdout Answer a FileStream that is attached the Smalltalk program's standard output file handle; this is used for normal output from the program.

6.69.3 FileStream: basic

copyFrom: from to: to
Answer the contents of the file between the two given positions

next Return the next character in the file, or nil at eof

nextByte Return the next byte in the file, or nil at eof

nextPut: aCharacter
Store aCharacter on the file

nextPutByte: anInteger
Store the byte, anInteger, on the file

nextPutByteArray: aByteArray
Store aByteArray on the file

position Answer the zero-based position from the start of the file

position: n
Set the file pointer to the zero-based position n

size Return the current size of the file, in bytes

truncate Truncate the file at the current position

6.69.4 FileStream: buffering

basicFlush Private - Flush the output buffer, fail if it is empty

bufferSize Answer the file's current buffer

bufferSize: bufSize
Flush the file and set the buffer's size to bufSize

clean Synchronize the file descriptor's state with the object's state.

fill Private - Fill the input buffer

flush Flush the output buffer

newBuffer Private - Answer a String to be used as the receiver's buffer

nextHunk Answer the next buffers worth of stuff in the Stream represented by the receiver. Do at most one actual input operation.

pendingWrite
Answer whether the output buffer is full

6.69.5 FileStream: filing in

fileIn File in the contents of the receiver. During a file in operation, global variables (starting with an uppercase letter) that are not declared don't yield an 'unknown variable' error. Instead, they are defined as nil in the 'Undeclared' dictionary (a global variable residing in Smalltalk). As soon as you add the variable to a namespace (for example by creating a class) the Association will be removed from Undeclared and reused in the namespace, so that the old references will automagically point to the new value.

fileInLine: lineNum fileName: aString at: charPosInt
Private - Much like a preprocessor #line directive; it is used by the Emacs Smalltalk mode.

6.69.6 FileStream: overriding inherited methods

next: anInteger
Return the next 'anInteger' characters from the stream, as a String.

nextByteArray: anInteger
Return the next 'anInteger' bytes from the stream, as a ByteArray.

nextPutAll: aCollection
Put all the characters in aCollection in the file

nextPutAllFlush: aCollection
Put all the characters in aCollection in the file, then flush the file buffers

6.69.7 FileStream: testing

atEnd Answer whether data has come to an end

6.70 Float

Defined in namespace Smalltalk
Category: Language-Data types

My instances represent floating point numbers that have 64 bits of precision (well, less than that in precision; they are precisely the same as C's "double" datatype). Besides the standard numerical operations, I provide transcendental operations too.

6.70.1 Float class: basic

e Returns the value of e. Hope is that it is precise enough

epsilon Return the smallest Float x for which is 1 + x ~= 1

infinity Return a Float that represents positive infinity. I hope that it is big enough, IEEE 8 byte floating point values (C doubles) overflow at 1e308.

largest Return the largest normalized Float that is not infinite.

ln10 Returns the value of ln 10. Hope is that it is precise enough

log10Base2
Returns the value of log2 10. Hope is that it is precise enough

mantissaBits
Answer the number of bits in the mantissa. 1 + (2^-mantissaBits) = 1

nan Return a Float that represents a mathematically indeterminate value (e.g. Inf - Inf, Inf / Inf)

negativeInfinity
Return a Float that represents negative infinity. I hope that it is big enough, IEEE 8 byte floating point values (C doubles) overflow at -1e308.

pi Returns the value of pi. Hope is that it is precise enough

smallest Return the smallest normalized Float that is not infinite.

smallestAbs
Return the smallest normalized Float that is > 0

6.70.2 Float class: byte-order dependancies

exponentByte
Answer the byte of the receiver that contains the exponent

leastSignificantMantissaByte
Answer the least significant byte in the receiver among those that contain the mantissa

6.70.3 Float class: converting

coerce: aNumber
: Answer aNumber converted to a Float

6.70.4 Float: arithmetic

// aNumber
: Return the integer quotient of dividing the receiver by aNumber with truncation towards negative infinity.

\\ aNumber
: Return the remainder of dividing the receiver by aNumber with truncation towards negative infinity.

integerPart
: Return the receiver's integer part

6.70.5 Float: built ins

*** arg** Multiply the receiver and arg and answer another Number

+ arg Sum the receiver and arg and answer another Number

- arg Subtract arg from the receiver and answer another Number

/ arg Divide the receiver by arg and answer another Float

< arg Answer whether the receiver is less than arg

<= arg Answer whether the receiver is less than or equal to arg

= arg Answer whether the receiver is equal to arg

> arg Answer whether the receiver is greater than arg

>= arg Answer whether the receiver is greater than or equal to arg

arcCos Answer the arc-cosine of the receiver

arcSin Answer the arc-sine of the receiver

arcTan Answer the arc-tangent of the receiver

ceiling Answer the integer part of the receiver, truncated towards +infinity

cos Answer the cosine of the receiver

exp Answer 'e' (2.718281828459...) raised to the receiver

exponent Answer the exponent of the receiver in mantissa*2^exponent representation (|mantissa|<=1)

floor Answer the integer part of the receiver, truncated towards -infinity

fractionPart
: Answer the fractional part of the receiver

hash Answer an hash value for the receiver

ln Answer the logarithm of the receiver in base 'e' (2.718281828459...)

primHash Private - Answer an hash value for the receiver

raisedTo: aNumber
Answer the receiver raised to its aNumber power

sin Answer the sine of the receiver

sqrt Answer the square root of the receiver

tan Answer the tangent of the receiver

timesTwoPower: arg
Answer the receiver multiplied by 2^arg

truncated Truncate the receiver towards zero and answer the result

~= arg Answer whether the receiver is not equal to arg

6.70.6 Float: coercing

asExactFraction
Convert the receiver into a fraction with optimal approximation, but with usually huge terms.

asFloat Just defined for completeness. Return the receiver.

asFraction Convert the receiver into a fraction with a good (but undefined) approximation

coerce: aNumber
Coerce aNumber to the receiver's class

estimatedLog
Answer an estimate of (self abs floorLog: 10)

generality Answer the receiver's generality

unity Coerce 1 to the receiver's class

zero Coerce 0 to the receiver's class

6.70.7 Float: printing

printOn: aStream
Print a representation of the receiver on aStream

6.70.8 Float: storing

storeOn: aStream
Print a representation of the receiver on aStream

6.70.9 Float: testing

isFinite Answer whether the receiver does not represent infinity, nor a NaN

isInfinite Answer whether the receiver represents positive or negative infinity

isNaN Answer whether the receiver represents a NaN

negative Answer whether the receiver is negative

positive Answer whether the receiver is positive

sign Answer 1 if the receiver is greater than 0, -1 if less than 0, else 0.

strictlyPositive
Answer whether the receiver is > 0

6.70.10 Float: testing functionality

isFloat Answer 'true'.

6.71 Fraction

Defined in namespace Smalltalk
Category: Language-Data types
I represent rational numbers in the form (p/q) where p and q are integers. The arithmetic operations *, +, -, /, on fractions, all return a reduced fraction.

6.71.1 Fraction class: converting

coerce: aNumber
Answer aNumber converted to a Fraction

6.71.2 Fraction class: instance creation

initialize Initialize the receiver's class variables

numerator: nInteger denominator: dInteger
Answer a new instance of fraction (nInteger/dInteger)

6.71.3 Fraction: accessing

denominator
Answer the receiver's denominator

numerator Answer the receiver's numerator

6.71.4 Fraction: arithmetic

*** aNumber**
Multiply two numbers and answer the result.

+ aNumber
Sum two numbers and answer the result.

- aNumber
Subtract aNumber from the receiver and answer the result.

/ aNumber
Divide the receiver by aNumber and answer the result.

// aNumber
Return the integer quotient of dividing the receiver by aNumber with truncation towards negative infinity.

\\ aNumber
Return the remainder from dividing the receiver by aNumber, (using //).

estimatedLog
Answer an estimate of (self abs floorLog: 10)

6.71.5 Fraction: coercing

coerce: aNumber
Coerce aNumber to the receiver's class

generality Return the receiver's generality

truncated Truncate the receiver and return the truncated result

unity Coerce 1 to the receiver's class

zero Coerce 0 to the receiver's class

6.71.6 Fraction: comparing

< arg Test if the receiver is less than arg.

<= arg Test if the receiver is less than or equal to arg.

= arg Test if the receiver equals arg.

> arg Test if the receiver is more than arg.

>= arg Test if the receiver is greater than or equal to arg.

hash Answer an hash value for the receiver

6.71.7 Fraction: converting

asFloat Answer the receiver converted to a Float

asFraction Answer the receiver converted to a Fraction

6.71.8 Fraction: optimized cases

negated Return the receiver, with its sign changed.

raisedToInteger: anInteger
Return self raised to the anInteger-th power.

reciprocal Return the reciprocal of the receiver.

squared Return the square of the receiver.

6.71.9 Fraction: printing

printOn: aStream
Print a representation of the receiver on aStream

storeOn: aStream
Store Smalltalk code compiling to the receiver on aStream

6.71.10 Fraction: testing

isRational Answer whether the receiver is rational - true

6.72 Halt

Defined in namespace Smalltalk
Category: Language-Exceptions
Halt represents a resumable error, usually a bug.

6.72.1 Halt: description

description
Answer a textual description of the exception.

isResumable
Answer true. #halt exceptions are by default resumable.

6.73 HashedCollection

Defined in namespace Smalltalk
Category: Collections-Unordered
I am an hashed collection that can store objects uniquely and give fast responses on their presence in the collection.

6.73.1 HashedCollection class: instance creation

new Answer a new instance of the receiver with a default size

new: anInteger
Answer a new instance of the receiver with the given size

6.73.2 HashedCollection: accessing

add: newObject
Add newObject to the set, if and only if the set doesn't already contain an occurrence of it. Don't fail if a duplicate is found. Answer anObject

at: index This method should not be called for instances of this class.

at: index put: value
This method should not be called for instances of this class.

6.73.3 HashedCollection: builtins

primAt: anIndex
Private - Answer the anIndex-th item of the hash table for the receiver. Using this instead of basicAt: allows for easier changes in the representation

primAt: anIndex put: value
Private - Store value in the anIndex-th item of the hash table for the receiver. Using this instead of basicAt:put: allows for easier changes in the representation

primSize Private - Answer the size of the hash table for the receiver. Using this instead of basicSize allows for easier changes in the representation

6.73.4 HashedCollection: copying

deepCopy Returns a deep copy of the receiver (the instance variables are copies of the receiver's instance variables)

shallowCopy
Returns a shallow copy of the receiver (the instance variables are not copied)

6.73.5 HashedCollection: enumerating the elements of a collection

do: aBlock
Enumerate all the non-nil members of the set

6.73.6 HashedCollection: rehashing

rehash Rehash the receiver

6.73.7 HashedCollection: Removing from a collection

remove: oldObject ifAbsent: anExceptionBlock
Remove oldObject to the set. If it is found, answer oldObject. Otherwise, evaluate anExceptionBlock and return its value.

6.73.8 HashedCollection: saving and loading

postLoad Called after loading an object; rehash the collection because identity objects will most likely mutate their hashes.

postStore Called after an object is dumped. Do nothing – necessary because by default this calls #postLoad by default

6.73.9 HashedCollection: storing

storeOn: aStream
Store on aStream some Smalltalk code which compiles to the receiver

6.73.10 HashedCollection: testing collections

= aHashedCollection
Returns true if the two sets have the same membership, false if not

capacity Answer how many elements the receiver can hold before having to grow.

hash Return the hash code for the members of the set. Since order is unimportant, we use a commutative operator to compute the hash value.

includes: anObject
Answer whether the receiver contains an instance of anObject.

isEmpty Answer whether the receiver is empty.

occurrencesOf: anObject
Return the number of occurrences of anObject. Since we're a set, this is either 0 or 1. Nil is never directly in the set, so we special case it (the result is always 1).

size Answer the receiver's size

6.74 IdentityDictionary

Defined in namespace Smalltalk
Category: Collections-Keyed
I am similar to an IdentityDictionary, except that removal and rehashing operations inside my instances look atomic to the interpreter.

6.75 IdentitySet

Defined in namespace Smalltalk
Category: Collections-Unordered
I am the typical set object; I can store any objects uniquely. I use the == operator to determine duplication of objects.

6.75.1 IdentitySet: testing

identityIncludes: anObject
Answer whether we include the anObject object; for IdentitySets this is identical to #includes:

6.76 Integer

Defined in namespace Smalltalk
Category: Language-Data types
I am the integer class of the GNU Smalltalk system. My instances can represent signed 30 bit integers and are as efficient as possible.

6.76.1 Integer class: converting

coerce: aNumber
Answer aNumber converted to a kind of Integer

6.76.2 Integer class: getting limits

bits Answer the number of bits (excluding the sign) that can be represented directly in an object pointer

largest Answer the largest integer represented directly in an object pointer

smallest Answer the smallest integer represented directly in an object pointer

6.76.3 Integer class: testing

isIdentity Answer whether x = y implies x == y for instances of the receiver

6.76.4 Integer: accessing

denominator
Answer '1'.

numerator Answer the receiver.

6.76.5 Integer: bit operators

allMask: anInteger
True if all 1 bits in anInteger are 1 in the receiver

anyMask: anInteger
True if any 1 bits in anInteger are 1 in the receiver

bitAt: index
Answer the index-th bit of the receiver (LSB: index = 1

bitClear: aMask
Answer an Integer equal to the receiver, except that all the bits that are set in aMask are cleared.

bitInvert Return the 1's complement of the bits of the receiver

clearBit: index
Clear the index-th bit of the receiver and answer a new Integer

highBit Return the index of the highest order 1 bit of the receiver

isBitSet: index
Answer whether the index-th bit of the receiver is set

noMask: anInteger
True if no 1 bits in anInteger are 1 in the receiver

setBit: index
Set the index-th bit of the receiver and answer a new Integer

6.76.6 Integer: Coercion methods (heh heh heh)

asCharacter
Return self as an ascii character

ceiling Return the receiver - it's already truncated

coerce: aNumber
Coerce aNumber to the receiver's class

floor Return the receiver - it's already truncated

generality Return the receiver's generality

rounded Return the receiver - it's already truncated

truncated Return the receiver - it's already truncated

unity Coerce 1 to the receiver's class

zero Coerce 0 to the receiver's class

6.76.7 Integer: converting

asFraction Return the receiver converted to a fraction

6.76.8 Integer: extension

alignTo: anInteger
Answer the receiver, truncated to the first higher or equal multiple of anInteger (which must be a power of two)

6.76.9 Integer: Math methods

estimatedLog
Answer an estimate of (self abs floorLog: 10)

even Return whether the receiver is even

factorial Return the receiver's factorial

floorLog: radix
return (self log: radix) floor. Optimized to answer an integer.

gcd: anInteger
Return the greatest common divisor (Euclid's algorithm) between the receiver and anInteger

lcm: anInteger
Return the least common multiple between the receiver and anInteger

odd Return whether the receiver is odd

6.76.10 Integer: Misc math operators

hash Answer an hash value for the receiver

6.76.11 Integer: Other iterators

timesRepeat: aBlock
Evaluate aBlock a number of times equal to the receiver's value. Compiled in-line for no argument aBlocks without temporaries, and therefore not overridable.

6.76.12 Integer: printing

printOn: aStream
Print on aStream the base 10 representation of the receiver

printOn: aStream base: b
Print on aStream the base b representation of the receiver

printString: baseInteger
Return the base b representation of the receiver

radix: baseInteger
Return the base b representation of the receiver, with BBr in front of it

storeOn: aStream base: b
Print on aStream Smalltalk code compiling to the receiver, represented in base b

6.76.13 Integer: storing

storeOn: aStream
Print on aStream the base 10 representation of the receiver

6.76.14 Integer: testing functionality

isInteger Answer 'true'.

isRational Answer whether the receiver is rational - true

isSmallInteger
Answer 'true'.

6.77 Interval

Defined in namespace Smalltalk
Category: Collections-Sequenceable
My instances represent ranges of objects, typically Number type objects. I provide iteration/enumeration messages for producing all the members that my instance represents.

6.77.1 Interval class: instance creation

from: startInteger to: stopInteger
Answer an Interval going from startInteger to the stopInteger, with a step of 1

from: startInteger to: stopInteger by: stepInteger
Answer an Interval going from startInteger to the stopInteger, with a step of stepInteger

withAll: aCollection
Answer an Interval containing the same elements as aCollection. Fail if it is not possible to create one.

6.77.2 Interval: basic

at: index Answer the index-th element of the receiver.

at: index put: anObject
This method should not be called for instances of this class.

collect: aBlock
Evaluate the receiver for each element in aBlock, collect in an array the result of the evaluations.

do: aBlock
Evaluate the receiver for each element in aBlock

reverse Answer a copy of the receiver with all of its items reversed

size Answer the number of elements in the receiver.

species Answer 'Array'.

6.77.3 Interval: printing

printOn: aStream
Print a representation for the receiver on aStream

6.77.4 Interval: storing

storeOn: aStream
Store Smalltalk code compiling to the receiver on aStream

6.77.5 Interval: testing

= anInterval
Answer whether anInterval is the same interval as the receiver

hash Answer an hash value for the receiver

6.78 LargeArray

Defined in namespace Smalltalk
Category: Collections-Sequenceable
I am similar to a plain array, but I'm specially designed to save memory when lots of items are nil.

6.78.1 LargeArray: overridden

newCollection: size
Create an Array of the given size

6.79 LargeArrayedCollection

Defined in namespace Smalltalk
Category: Collections-Sequenceable
I am an abstract class specially designed to save memory when lots of items have the same value.

6.79.1 LargeArrayedCollection class: instance creation

new: anInteger
Answer a new instance of the receiver, with room for anInteger elements.

6.79.2 LargeArrayedCollection: accessing

at: anIndex
Answer the anIndex-th item of the receiver.

at: anIndex put: anObject
Replace the anIndex-th item of the receiver with anObject.

compress Arrange the representation of the array for maximum memory saving.

6.79.3 LargeArrayedCollection: basic

= aLargeArray
Answer whether the receiver and aLargeArray have the same contents

hash Answer an hash value for the receiver

size Answer the maximum valid index for the receiver

6.80 LargeArraySubpart

Defined in namespace Smalltalk
Category: Collections-Sequenceable
This class is an auxiliary class used to store information about a LargeArrayedCollection's contents. LargeArrayedCollections store their items non-contiguously in a separate storage object, and use a SortedCollection to map between indices in the array and indices in the storage object; instances of this class represent a block of indices that is stored contiguously in the storage object.

6.80.1 LargeArraySubpart class: instance creation

first: first last: last index: index
Answer a LargeArraySubpart which answers first, last, and index when it is sent (respectively) #first, #last and #firstIndex.

6.80.2 LargeArraySubpart: accessing

first Answer the index of the first item of the LargeArrayedCollection that the receiver refers to.

first: firstIndex last: lastIndex index: storagePosition
Set up the receiver so that it answers first, last, and index when it is sent (respectively) #first, #last and #firstIndex.

firstIndex Answer the index in the collection's storage object of the first item of the LargeArrayedCollection that the receiver refers to.

last Answer the index of the last item of the LargeArrayedCollection that the receiver refers to.

lastIndex Answer the index in the collection's storage object of the last item of the LargeArrayedCollection that the receiver refers to.

6.80.3 LargeArraySubpart: comparing

< anObject
Answer whether the receiver points to a part of the array that is before anObject (this makes sense only if the receiver and anObject are two LargeArraySubparts referring to the same LargeArrayedCollection).

<= anObject
Answer whether the receiver points to a part of the array that is before anObject or starts at the same point (this makes sense only if the receiver and anObject are two LargeArraySubparts referring to the same LargeArrayedCollection).

= anObject
Answer whether the receiver and anObject are equal (assuming that the receiver and anObject are two LargeArraySubparts referring to the same LargeArrayed-Collection, which the receiver cannot check for).

hash Answer an hash value for the receiver

6.80.4 LargeArraySubpart: modifying

cutAt: position
Answer a new LargeArraySubpart whose lastIndex is position - 1, and apply a #removeFirst: to the receiver so that the firstIndex becomes position

grow Add one to last and lastIndex

growBy: numberOfElements
Add numberOfElements to last and lastIndex

relocateTo: position
Move the firstIndex to position, and the lastIndex accordingly.

removeFirst: n
Sum n to first and firstIndex, but leave last/lastIndex untouched

removeLast: n
Subtract n from last and lastIndex, but leave first/firstIndex untouched

6.81 LargeByteArray

Defined in namespace Smalltalk
Category: Collections-Sequenceable
I am similar to a plain ByteArray, but I'm specially designed to save memory when lots of items are zero.

6.81.1 LargeByteArray: overridden

costOfNewIndex
Answer the maximum number of consecutive items set to the defaultElement that can be present in a compressed array.

defaultElement
Answer the value which is hoped to be the most common in the array

newCollection: size
Create a ByteArray of the given size

6.82 LargeInteger

Defined in namespace Smalltalk
Category: Language-Data types
I represent a large integer, which has to be stored as a long sequence of bytes. I have methods to do arithmetics and comparisons, but I need some help from my children, LargePositiveInteger and LargeNegativeInteger, to speed them up a bit.

6.82.1 LargeInteger: arithmetic

*** aNumber**
Multiply aNumber and the receiver, answer the result

+ aNumber
Sum the receiver and aNumber, answer the result

- aNumber
Sum the receiver and aNumber, answer the result

/ aNumber
Divide aNumber and the receiver, answer the result (an Integer or Fraction)

// aNumber
Divide aNumber and the receiver, answer the result truncated towards -infinity

\\ aNumber
Divide aNumber and the receiver, answer the remainder truncated towards -infinity

estimatedLog
Answer an estimate of (self abs floorLog: 10)

negated Answer the receiver's negated

quo: aNumber
Divide aNumber and the receiver, answer the result truncated towards 0

rem: aNumber
Divide aNumber and the receiver, answer the remainder truncated towards 0

6.82.2 LargeInteger: bit operations

bitAnd: aNumber
Answer the receiver ANDed with aNumber

bitAt: aNumber
Answer the aNumber-th bit in the receiver, where the LSB is 1

bitInvert Answer the receiver's 1's complement

bitOr: aNumber
Answer the receiver ORed with aNumber

bitShift: aNumber
Answer the receiver shifted by aNumber places

bitXor: aNumber
Answer the receiver XORed with aNumber

6.82.3 LargeInteger: built-ins

at: anIndex
Answer the anIndex-th byte in the receiver's representation

at: anIndex put: aNumber
Answer the anIndex-th byte in the receiver's representation

digitAt: anIndex
Answer the anIndex-th base-256 digit in the receiver's representation

digitAt: anIndex put: aNumber
Answer the anIndex-th base-256 digit in the receiver's representation

digitLength
Answer the number of base-256 digits in the receiver

hash Answer an hash value for the receiver

primReplaceFrom: start to: stop with: replacementString
startingAt: replaceStart Private - Replace the characters from start to stop with new characters contained in replacementString (which, actually, can be any variable byte class, starting at the replaceStart location of replacementString

size Answer the number of indexed instance variable in the receiver

6.82.4 LargeInteger: coercion

coerce: aNumber
Truncate the number; if needed, convert it to LargeInteger representation.

generality Answer the receiver's generality

unity Coerce 1 to the receiver's class

zero Coerce 0 to the receiver's class

6.82.5 LargeInteger: disabled

asObject This method always fails. The number of OOPs is far less than the minimum number represented with a LargeInteger.

asObjectNoFail
Answer 'nil'.

6.82.6 LargeInteger: primitive operations

basicLeftShift: totalShift
Private - Left shift the receiver by aNumber places

basicRightShift: totalShift
Private - Right shift the receiver by 'shift' places

largeNegated
Private - Same as negated, but always answer a LargeInteger

6.82.7 LargeInteger: testing

< aNumber
Answer whether the receiver is smaller than aNumber

<= aNumber
Answer whether the receiver is smaller than aNumber or equal to it

= aNumber
Answer whether the receiver and aNumber identify the same number

> aNumber
Answer whether the receiver is greater than aNumber

>= aNumber
Answer whether the receiver is greater than aNumber or equal to it

~= aNumber
Answer whether the receiver and aNumber identify the same number

6.83 LargeNegativeInteger

Defined in namespace Smalltalk
Category: Language-Data types
Just like my brother LargePositiveInteger, I provide a few methods that allow LargeInteger to determine the sign of a large integer in a fast way during its calculations. For example, I know that I am smaller than any LargePositiveInteger

6.83.1 LargeNegativeInteger: converting

asFloat Answer the receiver converted to a Float

6.83.2 LargeNegativeInteger: numeric testing

abs Answer the receiver's absolute value.

negative Answer whether the receiver is < 0

positive Answer whether the receiver is >= 0

sign Answer the receiver's sign

strictlyPositive
Answer whether the receiver is > 0

6.83.3 LargeNegativeInteger: reverting to LargePositiveInteger

+ aNumber
Sum the receiver and aNumber, answer the result

- aNumber
Sum the receiver and aNumber, answer the result

gcd: anInteger
Return the greatest common divisor between the receiver and anInteger

highBit Answer the receiver's highest bit's index

6.84 LargePositiveInteger

Defined in namespace Smalltalk
Category: Language-Data types
Just like my brother LargeNegativeInteger, I provide a few methods that allow LargeInteger to determine the sign of a large integer in a fast way during its calculations. For example, I know that I am larger than any LargeNegativeInteger. In addition I implement the guts of arbitrary precision arithmetic.

6.84.1 LargePositiveInteger: arithmetic

+ aNumber
Sum the receiver and aNumber, answer the result

- aNumber
Subtract aNumber from the receiver, answer the result

gcd: anInteger
Calculate the GCD between the receiver and anInteger

highBit Answer the receiver's highest bit's index

6.84.2 LargePositiveInteger: converting

asFloat Answer the receiver converted to a Float

reverseStringBase: radix on: str
Return in a string the base radix representation of the receiver in reverse order

6.84.3 LargePositiveInteger: helper byte-level methods

bytes: byteArray1 from: j compare: byteArray2
Private - Answer the sign of byteArray2 - byteArray1; the j-th byte of byteArray1 is compared with the first of byteArray2, the j+1-th with the second, and so on.

bytes: byteArray1 from: j subtract: byteArray2
Private - Sutract the bytes in byteArray2 from those in byteArray1

bytes: bytes multiply: anInteger
Private - Multiply the bytes in bytes by anInteger, which must be < 255. Put the result back in bytes.

bytesLeftShift: aByteArray
Private - Left shift by 1 place the bytes in aByteArray

bytesLeftShift: aByteArray big: totalShift
Private - Left shift the bytes in aByteArray by totalShift places

bytesLeftShift: aByteArray n: shift
Private - Left shift by shift places the bytes in aByteArray (shift <= 7)

bytesRightShift: aByteArray big: totalShift
Private - Right shift the bytes in aByteArray by totalShift places

bytesRightShift: bytes n: aNumber
Private - Right shift the bytes in 'bytes' by 'aNumber' places (shift <= 7)

bytesTrailingZeros: bytes
Private - Answer the number of trailing zero bits in the receiver

primDivide: rhs
Private - Implements Knuth's divide and correct algorithm from 'Seminumerical Algorithms' 3rd Edition, section 4.3.1 (which is basically an enhanced version of the divide 'algorithm' for two-digit divisors which is taught in primary school!!!)

6.84.4 LargePositiveInteger: numeric testing

abs Answer the receiver's absolute value

negative Answer whether the receiver is < 0

positive Answer whether the receiver is >= 0

sign Answer the receiver's sign

strictlyPositive
Answer whether the receiver is > 0

6.84.5 LargePositiveInteger: primitive operations

divide: aNumber using: aBlock
Private - Divide the receiver by aNumber (unsigned division). Evaluate aBlock passing the result ByteArray, the remainder ByteArray, and whether the division had a remainder

isSmall Private - Answer whether the receiver is small enough to employ simple scalar algorithms for division and multiplication

multiply: aNumber
Private - Multiply the receiver by aNumber (unsigned multiply)

6.85 LargeWordArray

Defined in namespace Smalltalk
Category: Collections-Sequenceable
I am similar to a plain WordArray, but I'm specially designed to save memory when lots of items are zero.

6.85.1 LargeWordArray: overridden

defaultElement
Answer the value which is hoped to be the most common in the array

newCollection: size
Create a WordArray of the given size

6.86 LargeZeroInteger

Defined in namespace Smalltalk
Category: Language-Data types
I am quite a strange class. Indeed, the concept of a "large integer" that is zero is a weird one. Actually my only instance is zero but is represented like LargeIntegers, has the same generality as LargeIntegers, and so on. That only instance is stored in the class variable Zero, and is used in arithmetical methods, when we have to coerce a parameter that is zero.

6.86.1 LargeZeroInteger: accessing

at: anIndex
Answer '0'.

hash Answer '0'.

size Answer '0'.

6.86.2 LargeZeroInteger: arithmetic

*** aNumber**
Multiply aNumber and the receiver, answer the result

+ aNumber
Sum the receiver and aNumber, answer the result

- aNumber
Subtract aNumber from the receiver, answer the result

/ aNumber
Divide aNumber and the receiver, answer the result (an Integer or Fraction)

// aNumber
Divide aNumber and the receiver, answer the result truncated towards -infinity

\\ aNumber
Divide aNumber and the receiver, answer the remainder truncated towards -infinity

quo: aNumber
Divide aNumber and the receiver, answer the result truncated towards 0

rem: aNumber
Divide aNumber and the receiver, answer the remainder truncated towards 0

6.86.3 LargeZeroInteger: numeric testing

sign Answer the receiver's sign

strictlyPositive
Answer whether the receiver is > 0

6.86.4 LargeZeroInteger: printing

reverseStringBase: radix on: str
Return in a string the base radix representation of the receiver in reverse order

6.87 Link

Defined in namespace Smalltalk
Category: Collections-Sequenceable
I represent simple linked lists. Generally, I am not used by myself, but rather a subclass adds other instance variables that hold the information for each node, and I hold the glue that keeps them together.

6.87.1 Link class: instance creation

nextLink: aLink
Create an instance with the given next link

6.87.2 Link: basic

nextLink Answer the next item in the list

nextLink: aLink
Set the next item in the list

6.87.3 Link: iteration

at: index Retrieve a node (instance of Link) that is at a distance of 'index' after the receiver.

at: index put: object
This method should not be called for instances of this class.

do: aBlock
Evaluate aBlock for each element in the list

size Answer the number of elements in the list. Warning: this is O(n)

6.88 LinkedList

Defined in namespace Smalltalk
Category: Collections-Sequenceable
I provide methods that access and manipulate linked lists. I assume that the elements of the linked list are subclasses of Link, because I use the methods that class Link supplies to implement my methods.

6.88.1 LinkedList: accessing

at: index Return the element that is index into the linked list.

at: index put: object
This method should not be called for instances of this class.

6.88.2 LinkedList: adding

add: aLink
Add aLink at the end of the list; return aLink.

addFirst: aLink
Add aLink at the head of the list; return aLink.

addLast: aLink
Add aLink at then end of the list; return aLink.

remove: aLink ifAbsent: aBlock
Remove aLink from the list and return it, or invoke aBlock if it's not found in the list.

removeFirst
Remove the first element from the list and return it, or error if the list is empty.

removeLast
Remove the final element from the list and return it, or error if the list is empty.

6.88.3 LinkedList: enumerating

do: aBlock
Enumerate each object in the list, passing it to aBlock (actual behavior might depend on the subclass of Link that is being used).

6.88.4 LinkedList: testing

isEmpty Returns true if the list contains no members

notEmpty Returns true if the list contains at least a member

size Answer the number of elements in the list. Warning: this is O(n)

6.89 LookupKey

Defined in namespace Smalltalk
Category: Language-Data types
I represent a key for looking up entries in a data structure. Subclasses of me, such as Association, typically represent dictionary entries.

6.89.1 LookupKey class: basic

key: aKey Answer a new instance of the receiver with the given key and value

6.89.2 LookupKey: accessing

key Answer the receiver's key

key: aKey Set the receiver's key to aKey

6.89.3 LookupKey: printing

printOn: aStream
Put on aStream a representation of the receiver

6.89.4 LookupKey: storing

storeOn: aStream
Put on aStream some Smalltalk code compiling to the receiver

6.89.5 LookupKey: testing

< aLookupKey
Answer whether the receiver's key is less than aLookupKey's

= aLookupKey
Answer whether the receiver's key and value are the same as aLookupKey's, or false if aLookupKey is not an instance of the receiver

hash Answer an hash value for the receiver

6.90 LookupTable

Defined in namespace Smalltalk
Category: Collections-Keyed
I am similar to Dictionary, except that my representation is different (more efficient, but not as friendly to the virtual machine). I use the object equality comparision message = to determine equivalence of indices.

6.90.1 LookupTable class: instance creation

new Create a new LookupTable with a default size

6.90.2 LookupTable: accessing

add: anAssociation
Add the anAssociation key to the receiver

associationAt: key ifAbsent: aBlock
Answer the key/value Association for the given key. Evaluate aBlock (answering the result) if the key is not found

at: key ifAbsent: aBlock
Answer the value associated to the given key, or the result of evaluating aBlock if the key is not found

at: aKey ifPresent: aBlock
If aKey is absent, answer nil. Else, evaluate aBlock passing the associated value and answer the result of the invocation

at: key put: value
Store value as associated to the given key

6.90.3 LookupTable: copying

deepCopy Returns a deep copy of the receiver (the instance variables are copies of the receiver's instance variables)

6.90.4 LookupTable: enumerating

associationsDo: aBlock
Pass each association in the LookupTable to aBlock

keysAndValuesDo: aBlock
Pass each key/value pair in the LookupTable as two distinct parameters to aBlock

6.90.5 LookupTable: rehashing

rehash Rehash the receiver

6.90.6 LookupTable: removing

removeKey: key ifAbsent: aBlock
Remove the passed key from the LookupTable, answer the result of evaluating aBlock if it is not found

6.90.7 LookupTable: storing

storeOn: aStream
Print Smalltalk code compiling to the receiver on aStream

6.91 Magnitude

Defined in namespace Smalltalk
Category: Language-Data types
I am an abstract class. My objects represent things that are discrete and map to a number line. My instances can be compared with < and >.

6.91.1 Magnitude: basic

< aMagnitude
Answer whether the receiver is less than aMagnitude

<= aMagnitude
Answer whether the receiver is less than or equal to aMagnitude

= aMagnitude
Answer whether the receiver is equal to aMagnitude

> aMagnitude
Answer whether the receiver is greater than aMagnitude

>= aMagnitude
Answer whether the receiver is greater than or equal to aMagnitude

6.91.2 Magnitude: misc methods

between: min and: max
Returns true if object is inclusively between min and max.

max: aMagnitude
Returns the greatest object between the receiver and aMagnitude

min: aMagnitude
Returns the least object between the receiver and aMagnitude

6.92 MappedCollection

Defined in namespace Smalltalk
Category: Collections-Keyed

I represent collections of objects that are indirectly indexed by names. There are really two collections involved: domain and a map. The map maps between external names and indices into domain, which contains the real association. In order to work properly, the domain must be an instance of a subclass of SequenceableCollection, and the map must be an instance of Dictionary, or of a subclass of SequenceableCollection.

As an example of using me, consider implenting a Dictionary whose elements are indexed. The domain would be a SequenceableCollection with n elements, the map a Dictionary associating each key to an index in the domain. To access by key, to perform enumeration, etc. you would ask an instance of me; to access by index, you would access the domain directly.

Another idea could be to implement row access or column access to a matrix implemented as a single n*m Array: the Array would be the domain, while the map would be an Interval.

6.92.1 MappedCollection class: instance creation

collection: aCollection map: aMap
Answer a new MappedCollection using the given domain (aCollection) and map

new This method should not be used; instead, use #collection:map: to create MappedCollection.

6.92.2 MappedCollection: basic

add: anObject
This method should not be called for instances of this class.

at: key Answer the object at the given key

at: key put: value
Store value at the given key

collect: aBlock
Answer a MappedCollection with a copy of the receiver's map and a domain obtained by passing each object through aBlock

contents Answer a bag with the receiver's values

do: aBlock
Evaluate aBlock for each object

domain Answer the domain

map Answer the map

reject: aBlock
Answer the objects in the domain for which aBlock returns false

select: aBlock
Answer the objects in the domain for which aBlock returns true

size Answer the receiver's size

6.93 Memory

Defined in namespace Smalltalk
Category: Language-Implementation
I provide access to actual machine addresses of OOPs and objects. I have no instances; you send messages to my class to map between an object and the address of its OOP or object. In addition I provide direct memory access with different C types (ints, chars, OOPs, floats,...).

6.93.1 Memory class: accessing

at: anAddress
Access the Smalltalk object (OOP) at the given address.

at: anAddress put: aValue
Store a pointer (OOP) to the Smalltalk object identified by 'value' at the given address.

bigEndian Answer whether we're running on a big- or little-endian system.

charAt: anAddress
Access the C char at the given address. The value is returned as a Smalltalk Character.

charAt: anAddress put: aValue
Store as a C char the Smalltalk Character or Integer object identified by 'value', at the given address, using sizeof(char) bytes - i.e. 1 byte.

deref: anAddress
Access the C int pointed by the given address

doubleAt: anAddress
Access the C double at the given address.

doubleAt: anAddress put: aValue
Store the Smalltalk Float object identified by 'value', at the given address, writing it like a C double.

floatAt: anAddress
Access the C float at the given address.

floatAt: anAddress put: aValue
Store the Smalltalk Float object identified by 'value', at the given address, writing it like a C float.

intAt: anAddress
Access the C int at the given address.

intAt: anAddress put: aValue
Store the Smalltalk Integer object identified by 'value', at the given address, using sizeof(int) bytes.

longAt: anAddress
Access the C long int at the given address.

longAt: anAddress put: aValue
Store the Smalltalk Integer object identified by 'value', at the given address, using sizeof(long) bytes.

shortAt: anAddress
Access the C short int at the given address.

shortAt: anAddress put: aValue
Store the Smalltalk Integer object identified by 'value', at the given address, using sizeof(short) bytes.

stringAt: anAddress
Access the string pointed by the C 'char *' at the given given address.

stringAt: anAddress put: aValue
Store the Smalltalk String object identified by 'value', at the given address in memory, writing it like a *FRESHLY ALLOCATED* C string. It is the caller's responsibility to free it if necessary.

ucharAt: anAddress put: aValue
Store as a C char the Smalltalk Character or Integer object identified by 'value', at the given address, using sizeof(char) bytes - i.e. 1 byte.

uintAt: anAddress put: aValue
Store the Smalltalk Integer object identified by 'value', at the given address, using sizeof(int) bytes.

ulongAt: anAddress put: aValue
Store the Smalltalk Integer object identified by 'value', at the given address, using sizeof(long) bytes.

unsignedCharAt: anAddress
Access the C unsigned char at the given address. The value is returned as a Smalltalk Character.

unsignedCharAt: anAddress put: aValue
Store as a C char the Smalltalk Character or Integer object identified by 'value', at the given address, using sizeof(char) bytes - i.e. 1 byte.

unsignedIntAt: anAddress
Access the C unsigned int at the given address.

unsignedIntAt: anAddress put: aValue
Store the Smalltalk Integer object identified by 'value', at the given address, using sizeof(int) bytes.

unsignedLongAt: anAddress
Access the C unsigned long int at the given address.

unsignedLongAt: anAddress put: aValue
Store the Smalltalk Integer object identified by 'value', at the given address, using sizeof(long) bytes.

unsignedShortAt: anAddress
Access the C unsigned short int at the given address.

unsignedShortAt: anAddress put: aValue
Store the Smalltalk Integer object identified by 'value', at the given address, using sizeof(short) bytes.

ushortAt: anAddress put: aValue
Store the Smalltalk Integer object identified by 'value', at the given address, using sizeof(short) bytes.

6.93.2 Memory class: basic

addressOf: anObject
Returns the address of the actual object that anObject references. The result might be invalidated after a garbage collection occurs. Sending this method to Memory is deprecated; send it to ObjectMemory instead.

addressOfOOP: anObject
Returns the address of the OOP (object table slot) for anObject. The result is still valid after a garbage collection occurs. Sending this method to Memory is deprecated; send it to ObjectMemory instead.

type: aType at: anAddress
Returns a particular type object from memory at anAddress

type: aType at: anAddress put: aValue
Sets the memory location anAddress to aValue

6.94 Message

Defined in namespace Smalltalk
Category: Language-Implementation
I am a virtually existent class. By that I mean that logically instances of my class are created whenever a message is sent to an object, but in reality

my instances are only created to hold a message that has failed, so that error reporting methods can examine the sender and arguments.

6.94.1 Message class: creating instances

selector: aSymbol arguments: anArray
Create a new Message with the given selector and arguments

6.94.2 Message: accessing

argument Answer the first of the receiver's arguments

arguments Answer the receiver's arguments

arguments: anArray
Set the receiver's arguments

selector Answer the receiver's selector

selector: aSymbol
Set the receiver's selector

6.94.3 Message: basic

printOn: aStream
Print a representation of the receiver on aStream

reinvokeFor: aReceiver
Resend to aReceiver - present for compatibility

sendTo: aReceiver
Resend to aReceiver

6.95 MessageNotUnderstood

Defined in namespace Smalltalk
Category: Language-Exceptions
MessageNotUnderstood represents an error during message lookup. Signaling it is the default action of the #doesNotUnderstand: handler

6.95.1 MessageNotUnderstood: accessing

message Answer the message that wasn't understood

receiver Answer the object to whom the message send was directed

6.95.2 MessageNotUnderstood: description

description
Answer a textual description of the exception.

6.96 Metaclass

Defined in namespace Smalltalk
Category: Language-Implementation

I am the root of the class hierarchy. My instances are metaclasses, one for each real class. My instances have a single instance, which they hold onto, which is the class that they are the metaclass of. I provide methods for creation of actual class objects from metaclass object, and the creation of metaclass objects, which are my instances. If this is confusing to you, it should be...the Smalltalk metaclass system is strange and complex.

6.96.1 Metaclass class: instance creation

subclassOf: superMeta

Answer a new metaclass representing a subclass of superMeta

6.96.2 Metaclass: accessing

instanceClass

Answer the only instance of the metaclass

primaryInstance

Answer the only instance of the metaclass - present for compatibility

soleInstance

Answer the only instance of the metaclass - present for compatibility

6.96.3 Metaclass: basic

instanceVariableNames: classInstVarNames

Set the class-instance variables for the receiver to be those in classInstVarNames

name: newName

environment: aNamespace subclassOf: superclass instanceVariableNames: stringOfInstVarNames variable: variableBoolean words: wordBoolean pointers: pointerBoolean classVariableNames: stringOfClassVarNames poolDictionaries: stringOfPoolNames category: categoryName Private - create a full featured class and install it, or change an existing one

newMeta: className

environment: aNamespace subclassOf: superclass instanceVariableNames: stringOfInstVarNames variable: variableBoolean words: wordBoolean pointers: pointerBoolean classVariableNames: stringOfClassVarNames poolDictionaries: stringOfPoolNames category: categoryName Private - create a full featured class and install it

6.96.4 Metaclass: delegation

addClassVarName: aString
Add a class variable with the given name to the class pool dictionary

addSharedPool: aDictionary
Add the given shared pool to the list of the class' pool dictionaries

allClassVarNames
Answer the names of the variables in the receiver's class pool dictionary and in each of the superclasses' class pool dictionaries

allSharedPools
Return the names of the shared pools defined by the class and any of its superclasses

category Answer the class category

classPool Answer the class pool dictionary

classVarNames
Answer the names of the variables in the class pool dictionary

comment Answer the class comment

environment
Answer the namespace in which the receiver is implemented

name Answer the class name - it has none, actually

removeClassVarName: aString
Removes the class variable from the class, error if not present, or still in use.

removeSharedPool: aDictionary
Remove the given dictionary to the list of the class' pool dictionaries

sharedPools
Return the names of the shared pools defined by the class

6.96.5 Metaclass: filing

fileOutOn: aFileStream
File out complete class description: class definition, class and instance methods

6.96.6 Metaclass: printing

nameIn: aNamespace
Answer the class name when the class is referenced from aNamespace - a dummy one, since Behavior does not support names.

printOn: aStream
Print a represention of the receiver on aStream

storeOn: aStream
Store Smalltalk code compiling to the receiver on aStream

6.96.7 Metaclass: testing functionality

asClass Answer 'instanceClass'.

isBehavior Answer 'true'.

isMetaclass
Answer 'true'.

6.97 MethodContext

Defined in namespace Smalltalk
Category: Language-Implementation
My instances represent an actively executing method. They record various bits of information about the execution environment, and contain the execution stack.

6.97.1 MethodContext: accessing

home Answer the MethodContext to which the receiver refers (i.e. the receiver itself)

isBlock Answer whether the receiver is a block context

isEnvironment
To create a valid execution environment for the interpreter even before it starts, GST creates a fake context whose selector is nil and which can be used as a marker for the current execution environment. Answer whether the receiver is that kind of context.

sender Return the context from which the receiver was sent

6.97.2 MethodContext: printing

printOn: aStream
Print a representation for the receiver on aStream

6.98 MethodDictionary

Defined in namespace Smalltalk
Category: Language-Implementation

6.98.1 MethodDictionary: adding

at: key put: value
Store value as associated to the given key

6.98.2 MethodDictionary: rehashing

rehash Rehash the receiver

6.98.3 MethodDictionary: removing

removeAssociation: anAssociation
Remove anAssociation's key from the dictionary

removeKey: anElement ifAbsent: aBlock
Remove the passed key from the dictionary, answer the result of evaluating aBlock if it is not found

6.99 MethodInfo

Defined in namespace Smalltalk
Category: Language-Implementation
I provide information about particular methods. I can produce the category that a method was filed under, and can be used to access the source code of the method.

6.99.1 MethodInfo: accessing

category Answer the method category

category: aCategory
Set the method category

methodClass
Answer the class in which the method is defined

methodClass: aClass
Set the class in which the method is defined

selector Answer the selector through which the method is called

selector: aSymbol
Set the selector through which the method is called

sourceCode
Answer a FileSegment or String or nil containing the method source code

sourceFile Answer the name of the file where the method source code is

sourcePos Answer the starting position of the method source code in the sourceFile

sourceString
Answer a String containing the method source code

stripSourceCode
Remove the reference to the source code for the method

6.99.2 MethodInfo: equality

= aMethodInfo
Compare the receiver and aMethodInfo, answer whether they're equal

hash Answer an hash value for the receiver

6.100 Namespace

Defined in namespace Smalltalk
Category: Language-Implementation
I am a special form of dictionary. I provide special ways to access my keys, which typically begin with an uppercase letter. Classes hold on an instance of me; it is called their 'environment').

My keys are (expected to be) symbols, so I use == to match searched keys to those in the dictionary – this is done expecting it brings a bit more speed.

6.100.1 Namespace class: accessing

current Answer the current namespace

current: aNamespace
Set the current namespace to be aNamespace.

6.100.2 Namespace class: disabling instance creation

new Disabled - use #addSubspace: to create instances

new: size Disabled - use #addSubspace: to create instances

6.100.3 Namespace: accessing

inheritedKeys
Answer a Set of all the keys in the receiver and its superspaces

6.100.4 Namespace: namespace hierarchy

siblings Answer all the other namespaces that inherit from the receiver's superspace.

siblingsDo: aBlock
Evaluate aBlock once for each of the other namespaces that inherit from the receiver's superspace, passing the namespace as a parameter.

6.100.5 Namespace: overrides for superspaces

associationAt: key ifAbsent: aBlock
Return the key/value pair associated to the variable named as specified by 'key'. If the key is not found search will be brought on in superspaces, finally evaluating aBlock if the variable cannot be found in any of the superspaces.

associationsDo: aBlock
Pass each association in the namespace to aBlock

at: key ifAbsent: aBlock
Return the value associated to the variable named as specified by 'key'. If the key is not found search will be brought on in superspaces, finally evaluating aBlock if the variable cannot be found in any of the superspaces.

at: key ifPresent: aBlock
If aKey is absent from the receiver and all its superspaces, answer nil. Else, evaluate aBlock passing the associated value and answer the result of the invocation

do: aBlock
Pass each value in the namespace to aBlock

includesKey: key
Answer whether the receiver or any of its superspaces contain the given key

keysAndValuesDo: aBlock
Pass to aBlock each of the receiver's keys and values, in two separate parameters

keysDo: aBlock
Pass to aBlock each of the receiver's keys

set: key to: newValue ifAbsent: aBlock
Assign newValue to the variable named as specified by 'key'. This method won't define a new variable; instead if the key is not found it will search in superspaces and evaluate aBlock if it is not found. Answer newValue.

size Answer the number of keys in the receiver and each of its superspaces

6.100.6 Namespace: printing

name Answer the receiver's name

nameIn: aNamespace
Answer Smalltalk code compiling to the receiver when the current namespace is aNamespace

storeOn: aStream
Store Smalltalk code compiling to the receiver

6.100.7 Namespace: testing

isSmalltalk
Answer 'true'.

6.101 Notification

Defined in namespace Smalltalk
Category: Language-Exceptions

Notification represents a resumable, exceptional yet non-erroneous, situation. Signaling a notification in absence of an handler simply returns nil.

6.101.1 Notification: exception description

defaultAction

Do the default action for notifications, which is to resume execution of the context which signaled the exception.

description

Answer a textual description of the exception.

isResumable

Answer true. Notification exceptions are by default resumable.

6.102 NullProxy

Defined in namespace Smalltalk
Category: Streams-Files

I am a proxy that does no special processing on the object to be saved. I can be used to disable proxies for particular subclasses. My subclasses add to the stored information, but share the fact that the format is about the same as that of #dump: without a proxy.

6.102.1 NullProxy class: instance creation

loadFrom: anObjectDumper

Reload the object stored in anObjectDumper

6.102.2 NullProxy: accessing

dumpTo: anObjectDumper

Dump the object stored in the proxy to anObjectDumper

6.103 NullValueHolder

Defined in namespace Smalltalk
Category: Language-Data types

I pretend to store my value in a variable, but I don't actually. You can use the only instance of my class (returned by 'ValueHolder null') if you're not interested in a value that is returned as described in ValueHolder's comment.

6.103.1 NullValueHolder class: creating instances

new Not used – use 'ValueHolder null' instead

6.103.2 NullValueHolder: accessing

value Retrive the value of the receiver. Always answer nil

value: anObject
Set the value of the receiver. Do nothing, discard the value

6.104 Number

Defined in namespace Smalltalk
Category: Language-Data types
I am an abstract class that provides operations on numbers, both floating point and integer. I provide some generic predicates, and supply the implicit type coercing code for binary operations.

6.104.1 Number class: converting

coerce: aNumber
Answer aNumber - whatever class it belongs to, it is good

readFrom: aStream
Answer the number read from the rest of aStream, converted to an instance of the receiver. If the receiver is number, the class of the result is undefined – but the result is good.

6.104.2 Number class: testing

isImmediate
Answer whether, if x is an instance of the receiver, x copy == x

6.104.3 Number: arithmetic

*** aNumber**
Subtract the receiver and aNumber, answer the result

+ aNumber
Sum the receiver and aNumber, answer the result

- aNumber
Subtract aNumber from the receiver, answer the result

/ aNumber
Divide the receiver by aNumber, answer the result (no loss of precision)

// aNumber
Divide the receiver by aNumber, answer the result truncated towards -infinity

\\ aNumber
Divide the receiver by aNumber truncating towards -infinity, answer the remainder

quo: aNumber
Return the integer quotient of dividing the receiver by aNumber with truncation towards zero.

reciprocal Return the reciprocal of the receiver

rem: aNumber
Return the remainder of dividing the receiver by aNumber with truncation towards zero.

6.104.4 Number: converting

asFloat This method's functionality should be implemented by subclasses of Number

asFloatD This is mandated by the ANSI standard; since GNU Smalltalk currently does not support different floating-point classes, simply convert the receiver to a Float.

asFloatE This is mandated by the ANSI standard; since GNU Smalltalk currently does not support different floating-point classes, simply convert the receiver to a Float.

asFloatQ This is mandated by the ANSI standard; since GNU Smalltalk currently does not support different floating-point classes, simply convert the receiver to a Float.

asRectangle
Answer an empty rectangle whose origin is (self asPoint)

asScaledDecimal: n
Answer the receiver, converted to a ScaledDecimal object.

coerce: aNumber
Answer aNumber - whatever class it belongs to, it is good

degreesToRadians
Convert the receiver to radians

generality Answer the receiver's generality

radiansToDegrees
Convert the receiver from radians to degrees

unity Coerce 1 to the receiver's class. The default implementation works, but is inefficient

zero Coerce 0 to the receiver's class. The default implementation works, but is inefficient

6.104.5 Number: copying

deepCopy Return the receiver - it's an immediate (immutable) object

shallowCopy
Return the receiver - it's an immediate (immutable) object

6.104.6 Number: error raising

arithmeticError: msg
Raise an ArithmeticError exception having msg as its message text.

zeroDivide
Raise a division-by-zero (ZeroDivide) exception whose dividend is the receiver.

6.104.7 Number: Intervals & iterators

to: stop Return an interval going from the receiver to stop by 1

to: stop by: step
Return an interval going from the receiver to stop with the given step

to: stop by: step do: aBlock
Evaluate aBlock for each value in the interval going from the receiver to stop with the given step. Compiled in-line for integer literal steps, and for one-argument aBlocks without temporaries, and therefore not overridable.

to: stop do: aBlock
Evaluate aBlock for each value in the interval going from the receiver to stop by 1. Compiled in-line for one-argument aBlocks without temporaries, and therefore not overridable.

6.104.8 Number: misc math

abs Answer the absolute value of the receiver

arcCos return the arc cosine of the receiver

arcSin return the arc sine of the receiver

arcTan return the arc tangent of the receiver

cos return the cosine of the receiver

estimatedLog
Answer an estimate of (self abs floorLog: 10). This method should be overridden by subclasses, but Number's implementation does not raise errors - simply, it gives a correct result, so it is slow.

exp return e raised to the receiver

floorLog: radix
return (self log: radix) floor. Optimized to answer an integer.

ln return log base e of the receiver

log return log base aNumber of the receiver

log: aNumber
return log base aNumber of the receiver

negated Answer the negated of the receiver

raisedTo: aNumber
Return self raised to aNumber power

raisedToInteger: anInteger
Return self raised to the anInteger-th power

sin return the sine of the receiver

sqrt return the square root of the receiver

squared Answer the square of the receiver

tan return the tangent of the receiver

6.104.9 Number: point creation

@ y Answer a new point whose x is the receiver and whose y is y

asPoint Answer a new point, self @ self

6.104.10 Number: retrying

retry: aSymbol coercing: aNumber
Coerce to the other number's class the one number between the receiver and aNumber which has the lowest, and retry calling aSymbol. aSymbol is supposed not to be #= or #~= (since those don't fail if aNumber is not a Number).

retryDifferenceCoercing: aNumber
Coerce to the other number's class the one number between the receiver and aNumber which has the lowest, and retry calling #-.

retryDivisionCoercing: aNumber
Coerce to the other number's class the one number between the receiver and aNumber which has the lowest, and retry calling #/.

retryEqualityCoercing: aNumber
Coerce to the other number's class the one number between the receiver and aNumber which has the lowest, and retry calling #=.

retryError
Raise an error—a retrying method was called with two arguments having the same generality.

retryInequalityCoercing: aNumber
Coerce to the other number's class the one number between the receiver and aNumber which has the lowest, and retry calling #~=.

retryMultiplicationCoercing: aNumber
Coerce to the other number's class the one number between the receiver and aNumber which has the lowest, and retry calling #*.

retryRelationalOp: aSymbol coercing: aNumber
Coerce to the other number's class the one number between the receiver and aNumber which has the lowest, and retry calling aSymbol (<, <=, >, >=).

retrySumCoercing: aNumber
Coerce to the other number's class the one number between the receiver and aNumber which has the lowest, and retry calling #+.

6.104.11 Number: testing

closeTo: num
Answer whether the receiver can be considered sufficiently close to num (this is done by checking equality if num is not a number, and by checking with 0.01% tolerance if num is a number).

even Returns true if self is divisible by 2

isNumber Answer 'true'.

isRational Answer whether the receiver is rational - false by default

negative Answer whether the receiver is < 0

odd Returns true if self is not divisible by 2

positive Answer whether the receiver is >= 0

sign Returns the sign of the receiver.

strictlyPositive
Answer whether the receiver is > 0

6.104.12 Number: truncation and round off

asInteger Answer the receiver, rounded to the nearest integer

floor Return the integer nearest the receiver toward negative infinity.

fractionPart
Answer a number which, summed to the #integerPart of the receiver, gives the receiver itself.

integerPart
Answer the receiver, truncated towards zero

rounded Returns the integer nearest the receiver

roundTo: aNumber
Answer the receiver, truncated to the nearest multiple of aNumber

truncated Answer the receiver, truncated towards zero

truncateTo: aNumber
Answer the receiver, truncated towards zero to a multiple of aNumber

6.105 Object

Defined in namespace Smalltalk
Category: Language-Implementation
I am the root of the Smalltalk class system. All classes in the system are subclasses of me.

6.105.1 Object: built ins

= arg Answer whether the receiver is equal to arg. The equality test is by default the same as that for equal objects. = must not fail; answer false if the receiver cannot be compared to arg

== arg Answer whether the receiver is the same object as arg. This is a very fast test and is called 'identity'

addToBeFinalized
Add the object to the list of objects to be finalized when there are no more references to them

asOop Answer the object index associated to the receiver. The object index doesn't change when garbage collection is performed.

at: anIndex
Answer the index-th indexed instance variable of the receiver

at: anIndex put: value
Store value in the index-th indexed instance variable of the receiver

basicAt: anIndex
Answer the index-th indexed instance variable of the receiver. This method must not be overridden, override at: instead

basicAt: anIndex put: value
Store value in the index-th indexed instance variable of the receiver This method must not be overridden, override at:put: instead

basicPrint Print a basic representation of the receiver

basicSize Answer the number of indexed instance variable in the receiver

become: otherObject
Change all references to the receiver into references to otherObject. Depending on the implementation, references to otherObject might or might not be transformed into the receiver (respectively, 'two-way become' and 'one-way become').

Implementations doing one-way become answer the receiver (so that it is not lost). Most implementations doing two-way become answer otherObject, but this is not assured - so do answer the receiver for consistency. GNU Smalltalk does two-way become and answers otherObject, but this might change in future versions: programs should not rely on the behavior and results of #become: .

changeClassTo: aBehavior
Mutate the class of the receiver to be aBehavior. Note: Tacitly assumes that the structure is the same for the original and new class!!

checkIndexableBounds: index
Private - Check the reason why an access to the given indexed instance variable failed

checkIndexableBounds: index put: object
Private - Check the reason why a store to the given indexed instance variable failed

class Answer the class to which the receiver belongs

halt Called to enter the debugger

hash Answer an hash value for the receiver. This hash value is ok for objects that do not redefine ==.

identityHash
Answer an hash value for the receiver. This method must not be overridden

instVarAt: index
Answer the index-th instance variable of the receiver. This method must not be overridden.

instVarAt: index put: value
Store value in the index-th instance variable of the receiver. This method must not be overridden.

isReadOnly
Answer whether the object's indexed instance variables can be written

makeFixed
Avoid that the receiver moves in memory across garbage collections.

makeReadOnly: aBoolean
Set whether the object's indexed instance variables can be written

makeWeak
Make the object a 'weak' one. When an object is only referenced by weak objects, it is collected and the slots in the weak objects are changed to nils by the VM

mark: aSymbol
Private - use this method to mark code which needs to be reworked, removed, etc. You can then find all senders of #mark: to find all marked methods or you can look for all senders of the symbol that you sent to #mark: to find a category of marked methods.

nextInstance
Private - answer another instance of the receiver's class, or nil if the entire object table has been walked

notYetImplemented
Called when a method defined by a class is not yet implemented, but is going to be

perform: selectorOrMessageOrMethod
Send the unary message named selectorOrMessageOrMethod (if a Symbol) to the receiver, or the message and arguments it identifies (if a Message or DirectedMessage), or finally execute the method within the receiver (if a CompiledMethod). In the last case, the method need not reside on the hierarchy from the receiver's class to Object – it need not reside at all in a MethodDictionary, in fact – but doing bad things will compromise stability of the Smalltalk virtual machine (and don't blame anybody but yourself). This method should not be overridden

perform: selectorOrMethod with: arg1
Send the message named selectorOrMethod (if a Symbol) to the receiver, passing arg1 to it, or execute the method within the receiver (if a CompiledMethod). In the latter case, the method need not reside on the hierarchy from the receiver's class to Object – it need not reside at all in a MethodDictionary, in fact – but doing bad things will compromise stability of the Smalltalk virtual machine (and don't blame anybody but yourself). This method should not be overridden

perform: selectorOrMethod with: arg1 with: arg2
Send the message named selectorOrMethod (if a Symbol) to the receiver, passing arg1 and arg2 to it, or execute the method within the receiver (if a CompiledMethod). In the latter case, the method need not reside on the hierarchy from the receiver's class to Object – it need not reside at all in a MethodDictionary, in fact – but doing bad things will compromise stability of the Smalltalk virtual machine (and don't blame anybody but yourself). This method should not be overridden

perform: selectorOrMethod with: arg1 with: arg2 with: arg3
Send the message named selectorOrMethod (if a Symbol) to the receiver, passing the other arguments to it, or execute the method within the receiver (if a CompiledMethod). In the latter case, the method need not reside on the hierarchy from the receiver's class to Object – it need not reside at all in a MethodDictionary, in fact – but doing bad things will compromise stability of the Smalltalk virtual machine (and don't blame anybody but yourself). This method should not be overridden

perform: selectorOrMethod withArguments: argumentsArray
Send the message named selectorOrMethod (if a Symbol) to the receiver, passing the elements of argumentsArray as parameters, or execute the method within the receiver (if a CompiledMethod). In the latter case, the method need not reside on the hierarchy from the receiver's class to Object – it need

not reside at all in a MethodDictionary, in fact – but doing bad things will compromise stability of the Smalltalk virtual machine (and don't blame anybody but yourself). This method should not be overridden

primError: message
This might start the debugger... Note that we use #basicPrint 'cause #printOn: might invoke an error.

primitiveFailed
Called when a VM primitive fails

removeToBeFinalized
Remove the object from the list of objects to be finalized when there are no more references to them

shouldNotImplement
Called when objects belonging to a class should not answer a selector defined by a superclass

size Answer the number of indexed instance variable in the receiver

specialBasicAt: index
Similar to basicAt: but without bounds checking. This method is used to support instance mutation when an instance's class definition is changed. This method must not be overriddent

subclassResponsibility
Called when a method defined by a class should be overridden in a subclass

6.105.2 Object: change and update

broadcast: aSymbol
Send the unary message aSymbol to each of the receiver's dependents

broadcast: aSymbol with: anObject
Send the message aSymbol to each of the receiver's dependents, passing anObject

broadcast: aSymbol with: arg1 with: arg2
Send the message aSymbol to each of the receiver's dependents, passing arg1 and arg2 as parameters

broadcast: aSymbol withArguments: anArray
Send the message aSymbol to each of the receiver's dependents, passing the parameters in anArray

broadcast: aSymbol withBlock: aBlock
Send the message aSymbol to each of the receiver's dependents, passing the result of evaluating aBlock with each dependent as the parameter

changed Send update: for each of the receiver's dependents, passing them the receiver

changed: aParameter
Send update: for each of the receiver's dependents, passing them aParameter

update: aParameter
Default behavior is to do nothing. Called by #changed and #changed:

6.105.3 Object: class type methods

species This method has no unique definition. Generally speaking, methods which always return the same type usually don't use #class, but #species. For example, a PositionableStream's species is the class of the collection on which it is streaming (used by upTo:, upToAll:, upToEnd). Stream uses species for obtaining the class of next:'s return value, Collection uses it in its #copyEmpty: message, which in turn is used by all collection-re- turning methods. An Interval's species is Array (used by collect:, select:, reject:, etc.).

yourself Answer the receiver

6.105.4 Object: copying

copy Returns a shallow copy of the receiver (the instance variables are not copied). The shallow copy receives the message postCopy and the result of postCopy is passed back.

deepCopy Returns a deep copy of the receiver (the instance variables are copies of the receiver's instance variables)

postCopy Performs any changes required to do on a copied object. This is the place where one could, for example, put code to replace objects with copies of the objects

shallowCopy
Returns a shallow copy of the receiver (the instance variables are not copied)

6.105.5 Object: debugging

breakpoint: context return: return
Called back by the system. Must return the value passed through the second parameter

inspect Print all the instance variables of the receiver on the Transcript

validSize Answer how many elements in the receiver should be inspected

6.105.6 Object: dependents access

addDependent: anObject
Add anObject to the set of the receiver's dependents. Important: if an object has dependents, it won't be garbage collected.

dependents
Answer a collection of the receiver's dependents.

release Remove all of the receiver's dependents from the set and allow the receiver to be garbage collected.

removeDependent: anObject
Remove anObject to the set of the receiver's dependents. No problem if anObject is not in the set of the receiver's dependents.

6.105.7 Object: error raising

doesNotUnderstand: aMessage
Called by the system when a selector was not found. message is a Message containing information on the receiver

error: message
Display a walkback for the receiver, with the given error message. Signal an 'Error' exception (you can trap it the old way too, with 'ExError'

halt: message
Display a walkback for the receiver, with the given error message. Signal an 'Halt' exception (you can trap it the old way too, with 'ExHalt')

6.105.8 Object: finalization

finalize Do nothing by default

6.105.9 Object: printing

basicPrintNl
Print a basic representation of the receiver, followed by a new line.

basicPrintOn: aStream
Print a represention of the receiver on aStream

display Print a represention of the receiver on the Transcript (stdout the GUI is not active). For most objects this is simply its #print representation, but for strings and characters, superfluous dollars or extra pair of quotes are stripped.

displayNl Print a represention of the receiver, then put a new line on the Transcript (stdout the GUI is not active). For most objects this is simply its #printNl representation, but for strings and characters, superfluous dollars or extra pair of quotes are stripped.

displayOn: aStream
Print a represention of the receiver on aStream. For most objects this is simply its #printOn: representation, but for strings and characters, superfluous dollars or extra pair of quotes are stripped.

displayString
Answer a String representing the receiver. For most objects this is simply its #printString, but for strings and characters, superfluous dollars or extra pair of quotes are stripped.

print Print a represention of the receiver on the Transcript (stdout the GUI is not active)

printNl Print a represention of the receiver on stdout, put a new line the Transcript (stdout the GUI is not active)

printOn: aStream
Print a represention of the receiver on aStream

printString
Answer a String representing the receiver

6.105.10 Object: Relational operators

~= anObject
Answer whether the receiver and anObject are not equal

~~ anObject
Answer whether the receiver and anObject are not the same object

6.105.11 Object: saving and loading

binaryRepresentationObject
This method must be implemented if PluggableProxies are used with the receiver's class. The default implementation raises an exception.

postLoad Called after loading an object; must restore it to the state before 'preStore' was called. Do nothing by default

postStore Called after an object is dumped; must restore it to the state before 'preStore' was called. Call #postLoad by default

preStore Called before dumping an object; it must *change* it (it must not answer a new object) if necessary. Do nothing by default

reconstructOriginalObject
Used if an instance of the receiver's class is returned as the -#binaryRepresentationObject of another object. The default implementation raises an exception.

6.105.12 Object: storing

store Put a String of Smalltalk code compiling to the receiver on the Transcript (stdout the GUI is not active)

storeNl Put a String of Smalltalk code compiling to the receiver, followed by a new line, on the Transcript (stdout the GUI is not active)

storeOn: aStream
Put Smalltalk code compiling to the receiver on aStream

storeString
Answer a String of Smalltalk code compiling to the receiver

6.105.13 Object: syntax shortcuts

-> anObject
Creates a new instance of Association with the receiver being the key and the argument becoming the value

6.105.14 Object: testing functionality

ifNil: nilBlock
Evaluate nilBlock if the receiver is nil, else answer self

ifNil: nilBlock ifNotNil: notNilBlock
Evaluate nilBlock if the receiver is nil, else evaluate notNilBlock, passing the receiver.

ifNotNil: notNilBlock
Evaluate notNiilBlock if the receiver is not nil, passing the receiver. Else answer nil.

ifNotNil: notNilBlock ifNil: nilBlock
Evaluate nilBlock if the receiver is nil, else evaluate notNilBlock, passing the receiver.

isArray Answer 'false'.

isBehavior Answer 'false'.

isCharacter
Answer 'false'.

isCharacterArray
Answer 'false'.

isClass Answer 'false'.

isFloat Answer 'false'.

isInteger Answer 'false'.

isKindOf: aClass
Answer whether the receiver's class is aClass or a subclass of aClass

isMemberOf: aClass
Returns true if the receiver is an instance of the class 'aClass'

isMeta Same as isMetaclass

isMetaclass
Answer 'false'.

isMetaClass
Same as isMetaclass

isNamespace
Answer 'false'.

isNil Answer whether the receiver is nil

isNumber Answer 'false'.

isSmallInteger
Answer 'false'.

isString Answer 'false'.

isSymbol Answer 'false'.

notNil Answer whether the receiver is not nil

respondsTo: aSymbol
Returns true if the receiver understands the given selector

6.105.15 Object: VM callbacks

badReturnError
Called back when a block performs a bad return

mustBeBoolean
Called by the system when ifTrue:*, ifFalse:*, and: or or: are sent to anything but a boolean

noRunnableProcess
Called back when all processes are suspended

userInterrupt
Called back when the user presses Ctrl-Break

6.106 ObjectDumper

Defined in namespace Smalltalk
Category: Streams-Files
I'm not part of a normal Smalltalk system, but most Smalltalks provide a similar feature: that is, support for storing objects in a binary format; there are many advantages in using me instead of #storeOn: and the Smalltalk compiler.

The data is stored in a very compact format, which has the side effect of making loading much faster when compared with compiling the Smalltalk code prepared by #storeOn:. In addition, my instances support circular references between objects, while #storeOn: supports it only if you know of such references at design time and you override #storeOn: to deal with them

6.106.1 ObjectDumper class: establishing proxy classes

disableProxyFor: aClass
Disable proxies for instances of aClass and its descendants

hasProxyFor: aClass
Answer whether a proxy class has been registered for instances of aClass.

proxyClassFor: anObject
Answer the class of a valid proxy for an object, or nil if none could be found

proxyFor: anObject
Answer a valid proxy for an object, or the object itself if none could be found

registerProxyClass: aProxyClass for: aClass
Register the proxy class aProxyClass - descendent of DumperProxy - to be used for instances of aClass and its descendants

6.106.2 ObjectDumper class: instance creation

new This method should not be called for instances of this class.

on: aFileStream
Answer an ObjectDumper working on aFileStream.

6.106.3 ObjectDumper class: shortcuts

dump: anObject to: aFileStream
Dump anObject to aFileStream. Answer anObject

loadFrom: aFileStream
Load an object from aFileStream and answer it

6.106.4 ObjectDumper class: testing

example This is a real torture test: it outputs recursive objects, identical objects multiple times, classes, metaclasses, integers, characters and proxies (which is also a test of more complex objects)!

6.106.5 ObjectDumper: accessing

flush 'Forget' any information on previously stored objects.

stream Answer the ByteStream to which the ObjectDumper will write and from which it will read.

stream: aByteStream
Set the ByteStream to which the ObjectDumper will write and from which it will read.

6.106.6 ObjectDumper: loading/dumping objects

dump: anObject
Dump anObject on the stream associated with the receiver. Answer anObject

load Load an object from the stream associated with the receiver and answer it

6.106.7 ObjectDumper: stream interface

atEnd Answer whether the underlying stream is at EOF

next Load an object from the underlying stream

nextPut: anObject
Store an object on the underlying stream

6.107 ObjectMemory

Defined in namespace Smalltalk
Category: Language-Implementation
I provide a few methods that enable one to tune the virtual machine's usage of memory. In addition, I can signal to my dependants some 'events' that can happen during the virtual machine's life.

6.107.1 ObjectMemory class: builtins

addressOf: anObject
Returns the address of the actual object that anObject references. Note that, with the exception of fixed objects this address is only valid until the next garbage collection; thus it's pretty risky to count on the address returned by this method for very long.

addressOfOOP: anObject
Returns the address of the OOP (object table slot) for anObject. The address is an Integer and will not change over time (i.e. is immune from garbage collector action) except if the virtual machine is stopped and restarted.

compact Force a full garbage collection

gcMessage Answer whether messages indicating that garbage collection is taking place are printed on stdout

gcMessage: aBoolean
Set whether messages indicating that garbage collection is taking place are printed on stdout

growThresholdPercent
Answer the percentage of the amount of memory used by the system grows which has to be full for the system to allocate more memory

growThresholdPercent: growPercent
Set the percentage of the amount of memory used by the system grows which has to be full for the system to allocate more memory

growTo: numBytes
Grow the amount of memory used by the system grows to numBytes.

printStatistics
Print statistics about what the VM did since #resetStatistics was last called. Meaningful only if gst was made with 'make profile' or 'make profile_vm'

quit Quit the Smalltalk environment. Whether files are closed and other similar cleanup occurs depends on the platform

quit: exitStatus
Quit the Smalltalk environment, passing the exitStatus integer to the OS. Files are closed and other similar cleanups occur.

resetStatistics
Reset the statistics about the VM which #printStatistics can print.

snapshot: aString
Save an image on the aString file

spaceGrowRate
Answer the rate with which the amount of memory used by the system grows

spaceGrowRate: rate
Set the rate with which the amount of memory used by the system grows

6.107.2 ObjectMemory class: dependancy

update: aspect
Fire the init blocks for compatibility with previous versions

6.107.3 ObjectMemory class: initialization

initialize Initialize the globals

6.107.4 ObjectMemory class: saving the image

snapshot Save a snapshot on the image file that was loaded on startup.

6.108 OrderedCollection

Defined in namespace Smalltalk
Category: Collections-Sequenceable
My instances represent ordered collections of arbitrary typed objects which are not directly accessible by an index. They can be accessed indirectly through an index, and can be manipulated by adding to the end or based on content (such as add:after:)

6.108.1 OrderedCollection class: instance creation

new Answer an OrderedCollection of default size

new: anInteger
Answer an OrderedCollection of size anInteger

6.108.2 OrderedCollection: accessing

at: anIndex
Answer the anIndex-th item of the receiver

at: anIndex put: anObject
Store anObject at the anIndex-th item of the receiver, answer anObject

size Return the number of objects in the receiver

6.108.3 OrderedCollection: adding

add: anObject
Add anObject in the receiver, answer it

add: newObject after: oldObject
Add newObject in the receiver just after oldObject, answer it. Fail if oldObject can't be found

add: newObject afterIndex: i
Add newObject in the receiver just after the i-th, answer it. Fail if i < 0 or i > self size

add: newObject before: oldObject
Add newObject in the receiver just before oldObject, answer it. Fail if oldObject can't be found

add: newObject beforeIndex: i
Add newObject in the receiver just before the i-th, answer it. Fail if i < 1 or i > self size + 1

addAll: aCollection
Add every item of aCollection to the receiver, answer it

addAll: newCollection after: oldObject
Add every item of newCollection to the receiver just after oldObject, answer it. Fail if oldObject is not found

addAll: newCollection afterIndex: i
Add every item of newCollection to the receiver just after the i-th, answer it. Fail if i < 0 or i > self size

addAll: newCollection before: oldObject
Add every item of newCollection to the receiver just before oldObject, answer it. Fail if oldObject is not found

addAll: newCollection beforeIndex: i
Add every item of newCollection to the receiver just before the i-th, answer it. Fail if i < 1 or i > self size + 1

addAllFirst: aCollection
Add every item of newCollection to the receiver right at the start of the receiver. Answer aCollection

addAllLast: aCollection
Add every item of newCollection to the receiver right at the end of the receiver. Answer aCollection

addFirst: newObject
Add newObject to the receiver right at the start of the receiver. Answer newObject

addLast: newObject
Add newObject to the receiver right at the end of the receiver. Answer newObject

6.108.4 OrderedCollection: removing

remove: anObject ifAbsent: aBlock
Remove anObject from the receiver. If it can't be found, answer the result of evaluating aBlock

removeAtIndex: anIndex
Remove the object at index anIndex from the receiver. Fail if the index is out of bounds

removeFirst
Remove an object from the start of the receiver. Fail if the receiver is empty

removeLast
Remove an object from the end of the receiver. Fail if the receiver is empty

6.109 PackageLoader

Defined in namespace Smalltalk
Category: Language-Data types
I am not part of a standard Smalltalk system. I provide methods for loading packages into a Smalltalk image, correctly handling dependencies.

6.109.1 PackageLoader class: accessing

addPackage: package directory: dir fileIn: fileIns needs: prerequisites
Add the given package to the 'packages' file, with the given directory (if relative, it is relative to the kernel directory), fileIns and prerequisites. fileIns and prerequisites should be two Collections of Strings. Note that none of this fields are optional. If there are no prere- quisites, just use #('Kernel') as the prerequisites.

directoryFor: package
Answer a complete path to the given package's file-in

fileInsFor: package
Answer a Set of Strings containing the filenames of the given package's file-ins (relative to the directory answered by #directoryFor:)

filesFor: package
Answer a Set of Strings containing the filenames of the given package's files (file-ins are relative to the directory answered by #directoryFor:, shared modules are relative to the ModulePath)

ignoreCallouts
Answer whether unavailable C callouts must generate errors or not.

ignoreCallouts: aBoolean
Set whether unavailable C callouts must generate errors or not.

modulesFor: package
Answer a Set of Strings containing the filenames of the given package's file-ins (relative to the directory answered by #directoryFor:)

prerequisitesFor: package
Answer a Set of Strings containing the prerequisites for the given package

refreshDependencies
Reload the 'packages' file in the image directory

6.109.2 PackageLoader class: loading

extractDependenciesFor: packagesList onError: aBlock
Answer an OrderedCollection containing all the packages which you have to load to enable the packages in packagesList, in an appropriate order. For example PackageLoader extractDependenciesFor: #('BloxTestSuite' 'Blox' 'Browser') on a newly built image will evaluate to an OrderedCollection containing 'Kernel', 'C:tclInit', 'Blox', 'BloxTestSuite' and 'Browser'. Note that Blox has been moved before BloxTestSuite. Pass an error message to aBlock if any of the packages needs C call-outs which are not defined.

fileInPackage: package
File in the given package into GNU Smalltalk.

fileInPackages: packagesList
File in all the packages in packagesList into GNU Smalltalk.

6.109.3 PackageLoader class: testing

canLoad: package
Answer whether all the needed C call-outs are registered within GNU Smalltalk

6.110 PluggableAdaptor

Defined in namespace Smalltalk
Category: Language-Data types
I mediate between complex get/set behavior and the #value/#value: protocol used by ValueAdaptors. The get/set behavior can be implemented by two blocks, or can be delegated to another object with messages such as -#someProperty to get and #someProperty: to set.

6.110.1 PluggableAdaptor class: creating instances

getBlock: getBlock putBlock: putBlock
Answer a PluggableAdaptor using the given blocks to implement #value and #value:

on: anObject aspect: aSymbol
Answer a PluggableAdaptor using anObject's aSymbol message to implement #value, and anObject's aSymbol: message (aSymbol followed by a colon) to implement #value:

on: anObject getSelector: getSelector putSelector: putSelector
Answer a PluggableAdaptor using anObject's getSelector message to implement #value, and anObject's putSelector message to implement #value:

on: anObject index: anIndex
Answer a PluggableAdaptor using anObject's #at: and #at:put: message to implement #value and #value:; the first parameter of #at: and #at:put: is anIndex

on: aDictionary key: aKey
Same as #on:index:. Provided for clarity and completeness.

6.110.2 PluggableAdaptor: accessing

value Get the value of the receiver.

value: anObject
Set the value of the receiver.

6.111 PluggableProxy

Defined in namespace Smalltalk
Category: Streams-Files
I am a proxy that stores a different object and, upon load, sends #reconstructOriginalObject to that object (which can be a DirectedMessage, in which case the message is sent). The object to be stored is retrieved by sending #binaryRepresentationObject to the object.

6.111.1 PluggableProxy class: accessing

on: anObject
Answer a proxy to be used to save anObject. The proxy stores a different object obtained by sending to anObject the #binaryRepresentationObject message (embedded between #preStore and #postStore as usual).

6.111.2 PluggableProxy: saving and restoring

object Reconstruct the object stored in the proxy and answer it; the binaryRepresentationObject is sent the #reconstructOriginalObject message, and the resulting object is sent the #postLoad message.

6.112 Point

Defined in namespace Smalltalk
Category: Language-Data types
Beginning of a Point class for simple display manipulation. Has not been exhaustively tested but appears to work for the basic primitives and for the needs of the Rectangle class.

6.112.1 Point class: instance creation

new Create a new point with both coordinates set to 0

x: xInteger y: yInteger
Create a new point with the given coordinates

6.112.2 Point: accessing

x Answer the x coordinate

x: aNumber
Set the x coordinate to aNumber

x: anXNumber y: aYNumber
Set the x and y coordinate to anXNumber and aYNumber, respectively

y Answer the y coordinate

y: aNumber
Set the y coordinate to aNumber

6.112.3 Point: arithmetic

*** scale** Multiply the receiver by scale, which can be a Number or a Point

+ delta Sum the receiver and delta, which can be a Number or a Point

- delta Subtract delta, which can be a Number or a Point, from the receiver

/ scale Divide the receiver by scale, which can be a Number or a Point, with no loss of precision

// scale Divide the receiver by scale, which can be a Number or a Point, with truncation towards -infinity

abs Answer a new point whose coordinates are the absolute values of the receiver's

6.112.4 Point: comparing

< aPoint Answer whether the receiver is higher and to the left of aPoint

<= aPoint Answer whether aPoint is equal to the receiver, or the receiver is higher and to the left of aPoint

= aPoint Answer whether the receiver is equal to aPoint

> aPoint Answer whether the receiver is lower and to the right of aPoint

>= aPoint Answer whether aPoint is equal to the receiver, or the receiver is lower and to the right of aPoint

max: aPoint
Answer self if it is lower and to the right of aPoint, aPoint otherwise

min: aPoint
Answer self if it is higher and to the left of aPoint, aPoint otherwise

6.112.5 Point: converting

asPoint Answer the receiver.

asRectangle
Answer an empty rectangle whose origin is self

corner: aPoint
Answer a Rectangle whose origin is the receiver and whose corner is aPoint

extent: aPoint
Answer a Rectangle whose origin is the receiver and whose extent is aPoint

hash Answer an hash value for the receiver

6.112.6 Point: point functions

arcTan Answer the angle (measured counterclockwise) between the receiver and a ray starting in (0, 0) and moving towards (1, 0) - i.e. 3 o'clock

dist: aPoint
Answer the distance between the receiver and aPoint

dotProduct: aPoint
Answer the dot product between the receiver and aPoint

grid: aPoint
Answer a new point whose coordinates are rounded towards the nearest multiple of aPoint

normal Rotate the Point 90degrees clockwise and get the unit vector

transpose Answer a new point whose coordinates are the receiver's coordinates exchanged (x becomes y, y becomes x)

truncatedGrid: aPoint
Answer a new point whose coordinates are rounded towards -infinity, to a multiple of grid (which must be a Point)

6.112.7 Point: printing

printOn: aStream
Print a representation for the receiver on aStream

6.112.8 Point: storing

storeOn: aStream
Print Smalltalk code compiling to the receiver on aStream

6.112.9 Point: truncation and round off

rounded Answer a new point whose coordinates are rounded to the nearest integer

truncateTo: grid
Answer a new point whose coordinates are rounded towards -infinity, to a multiple of grid (which must be a Number)

6.113 PositionableStream

Defined in namespace Smalltalk
Category: Streams-Collections
My instances represent streams where explicit positioning is permitted. Thus, my streams act in a manner to normal disk files: you can read or write sequentially, but also position the file to a particular place whenever you choose. Generally, you'll want to use ReadStream, WriteStream or ReadWriteStream instead of me to create and use streams.

6.113.1 PositionableStream class: instance creation

on: aCollection
Answer an instance of the receiver streaming on the whole contents of aCollection

on: aCollection from: firstIndex to: lastIndex
Answer an instance of the receiver streaming from the firstIndex-th item of aCollection to the lastIndex-th

6.113.2 PositionableStream: accessing-reading

close Disassociate a stream from its backing store.

contents Returns a collection of the same type that the stream accesses, up to and including the final element.

copyFrom: start to: end
Answer the collection on which the receiver is streaming, from the start-th item to the end-th

next Answer the next item of the receiver

peek Returns the next element of the stream without moving the pointer. Returns nil when at end of stream.

peekFor: anObject
Returns true and gobbles the next element from the stream of it is equal to anObject, returns false and doesn't gobble the next element if the next element is not equal to anObject.

reverseContents
Returns a collection of the same type that the stream accesses, up to and including the final element, but in reverse order.

6.113.3 PositionableStream: class type methods

isExternalStream
We stream on a collection residing in the image, so answer false

species The collections returned by #upTo: etc. are the same kind as those returned by the collection with methods such as #select:

6.113.4 PositionableStream: positioning

basicPosition: anInteger
Move the stream pointer to the anInteger-th object

position Answer the current value of the stream pointer

position: anInteger
Move the stream pointer to the anInteger-th object

reset Move the stream back to its first element. For write-only streams, the stream is truncated there.

setToEnd Move the current position to the end of the stream.

size Answer the size of data on which we are streaming.

skip: anInteger
Move the current position by anInteger places, either forwards or backwards.

skipSeparators
Advance the receiver until we find a character that is not a separator. Answer false if we reach the end of the stream, else answer true; in this case, sending #next will return the first non-separator character (possibly the same to which the stream pointed before #skipSeparators was sent).

6.113.5 PositionableStream: testing

atEnd Answer whether the objects in the stream have reached an end

basicAtEnd
Answer whether the objects in the stream have reached an end. This method must NOT be overridden.

isEmpty Answer whether the stream has no objects

6.113.6 PositionableStream: truncating

truncate Truncate the receiver to the current position - only valid for writing streams

6.114 Process

Defined in namespace Smalltalk
Category: Language-Processes
I represent a unit of computation. My instances are independantly executable blocks that have a priority associated with them, and they can suspend themselves and resume themselves however they wish.

6.114.1 Process class: basic

on: aBlockContext at: aPriority
Private - Create a process running aBlockContext at the given priority

6.114.2 Process: accessing

name Answer 'name'.

name: aString
Give the name aString to the process

priority Answer the receiver's priority

priority: anInteger
Change the receiver's priority to anInteger

queueInterrupt: aBlock
Force the receiver to be interrupted and to evaluate aBlock as soon as it becomes the active process (this could mean NOW if the receiver is active). Answer the receiver

6.114.3 Process: basic

forceResume
Private - Force a resume of the process from whatever status it was in (even if it was waiting on a semaphore). This is BAD practice, it is present only for some future possibility.

lowerPriority
Lower a bit the priority of the receiver. A #lowerPriority will cancel a previous #raisePriority, and vice versa.

raisePriority
Raise a bit the priority of the receiver. A #lowerPriority will cancel a previous #raisePriority, and vice versa.

suspend Do nothing if we're already suspended. Note that the blue book made suspend a primitive - but the real primitive is yielding control to another process. Suspending is nothing more than taking ourselves out of every scheduling list and THEN yield control to another process

terminate Terminate the receiver - This is nothing more than prohibiting to resume the process, then suspending it.

6.114.4 Process: builtins

resume Resume the receiver's execution

yield Yield control from the receiver to other processes

6.114.5 Process: printing

printOn: aStream
Print a representation of the receiver on aStream

6.115 ProcessorScheduler

Defined in namespace Smalltalk
Category: Language-Processes
I provide methods that control the execution of processes.

6.115.1 ProcessorScheduler class: instance creation

new Error—new instances of ProcessorScheduler should not be created.

6.115.2 ProcessorScheduler: basic

activePriority
Answer the active process' priority

activeProcess
Answer the active process

changePriorityOf: aProcess to: aPriority
Private - Move aProcess to the execution list for aPriority, answer the new execution list

processesAt: aPriority
Private - Answer a linked list of processes at the given priority

terminateActive
Private - Terminate the active process

yield Let the active process yield control to other processes

6.115.3 ProcessorScheduler: idle tasks

idle Private - Call the next idle task

idleAdd: aBlock
Register aBlock to be executed when things are idle

6.115.4 ProcessorScheduler: printing

printOn: aStream
Store onto aStream a printed representation of the receiver

6.115.5 ProcessorScheduler: priorities

highestPriority
Answer the highest valid priority

highIOPriority
Answer the priority for system high-priority I/O processes, such as a process handling input from a network.

lowestPriority
Answer the lowest valid priority

lowIOPriority
Answer the priority for system low-priority I/O processes. Examples are the process handling input from the user (keyboard, pointing device, etc.) and the process distributing input from a network.

priorityName: priority
Private - Answer a name for the given process priority

rockBottomPriority
Answer the lowest valid priority

systemBackgroundPriority
Answer the priority for system background-priority processes. Examples are an incremental garbage collector or status checker.

timingPriority
Answer the priority for system real-time processes.

unpreemptedPriority
Answer the highest priority avilable in the system; never create a process with this priority, instead use BlockClosure>>#valueWithoutPreemption.

userBackgroundPriority
Answer the priority for user background-priority processes

userInterruptPriority
Answer the priority for user interrupt-priority processes. Processes run at this level will preempt the window scheduler and should, therefore, not consume the processor forever.

userSchedulingPriority
Answer the priority for user standard-priority processes

6.115.6 ProcessorScheduler: storing

storeOn: aStream
Store onto aStream a Smalltalk expression which evaluates to the receiver

6.115.7 ProcessorScheduler: timed invocation

isTimeoutProgrammed
Private - Answer whether there is a pending call to #signal:atMilliseconds:

signal: aSemaphore atMilliseconds: millis
Private - signal 'aSemaphore' after 'millis' milliseconds have elapsed

signal: aSemaphore onInterrupt: anIntegerSignalNumber
Private - signal 'aSemaphore' when the given C signal occurs

6.116 Promise

Defined in namespace Smalltalk
Category: Language-Data types

6.116.1 Promise class: creating instances

null This method should not be called for instances of this class.

6.116.2 Promise: accessing

hasValue Answer whether we already have a value.

value Get the value of the receiver.

value: anObject
Set the value of the receiver.

6.116.3 Promise: initializing

initialize Private - set the initial state of the receiver

6.117 Random

Defined in namespace Smalltalk
Category: Streams
My instances are generator streams that produce random numbers, which are floating point values between 0 and 1.

6.117.1 Random class: instance creation

new Create a new random number generator whose seed is given by the current time on the millisecond clock

seed: aFloat
Create a new random number generator whose seed is aFloat

6.117.2 Random: basic

atEnd This stream never ends. Always answer false

next Return the next random number in the sequence

nextPut: value
This method should not be called for instances of this class.

6.117.3 Random: testing

chiSquare returns under Pentium II, NT 4.0, 93.0

chiSquare: n range: r
Return the chi-square deduced from calculating n random numbers in the 0..r range

6.118 ReadStream

Defined in namespace Smalltalk
Category: Streams-Collections

I implement the set of read-only stream objects. You may read from my objects, but you may not write to them.

6.118.1 ReadStream class: instance creation

on: aCollection

Answer a new stream working on aCollection from its start.

6.118.2 ReadStream: accessing-reading

reverseContents

May be faster than generic stream reverseContents.

size Answer the receiver's size.

6.119 ReadWriteStream

Defined in namespace Smalltalk
Category: Streams-Collections

I am the class of streams that may be read and written from simultaneously. In some sense, I am the best of both ReadStream and WriteStream.

6.119.1 ReadWriteStream class: instance creation

on: aCollection

Answer a new stream working on aCollection from its start. The stream starts at the front of aCollection

with: aCollection

Answer a new instance of the receiver which streams from the end of aCollection.

6.119.2 ReadWriteStream: positioning

position: anInteger

Unlike WriteStreams, ReadWriteStreams don't truncate the stream

skip: anInteger

Unlike WriteStreams, ReadWriteStreams don't truncate the stream

6.120 Rectangle

Defined in namespace Smalltalk
Category: Language-Data types

Beginning of the Rectangle class for simple display manipulation. Rectangles require the Point class to be available. An extension to the Point class is made here that since it requires Rectangles to be defined (see converting)

6.120.1 Rectangle class: instance creation

left: leftNumber right: rightNumber top: topNumber bottom: bottomNumber
Answer a rectangle with the given coordinates

new Answer the (0 @ 0 corner: 0 @ 0) rectangle

origin: originPoint corner: cornerPoint
Answer a rectangle with the given corners

origin: originPoint extent: extentPoint
Answer a rectangle with the given origin and size

6.120.2 Rectangle: accessing

bottom Answer the corner's y of the receiver

bottom: aNumber
Set the corner's y of the receiver

bottomCenter
Answer the center of the receiver's bottom side

bottomLeft
Answer the bottom-left corner of the receiver

bottomLeft: aPoint
Answer the receiver with the bottom-left changed to aPoint

bottomRight
Answer the bottom-right corner of the receiver

bottomRight: aPoint
Change the bottom-right corner of the receiver

center Answer the center of the receiver

corner Answer the corner of the receiver

corner: aPoint
Set the corner of the receiver

extent Answer the extent of the receiver

extent: aPoint
Change the size of the receiver, keeping the origin the same

height Answer the height of the receiver

height: aNumber
Set the height of the receiver

left Answer the x of the left edge of the receiver

left: aValue
Set the x of the left edge of the receiver

left: l top: t right: r bottom: b
Change all four the coordinates of the receiver's corners

leftCenter Answer the center of the receiver's left side

origin Answer the top-left corner of the receiver

origin: aPoint
Change the top-left corner of the receiver to aPoint

origin: pnt1 corner: pnt2
Change both the origin (top-left corner) and the corner (bottom-right corner) of the receiver

origin: pnt1 extent: pnt2
Change the top-left corner and the size of the receiver

right Answer the x of the bottom-right corner of the receiver

right: aNumber
Change the x of the bottom-right corner of the receiver

rightCenter
Answer the center of the receiver's right side

top Answer the y of the receiver's top-left corner

top: aValue
Change the y of the receiver's top-left corner

topCenter Answer the center of the receiver's top side

topLeft Answer the receiver's top-left corner

topLeft: aPoint
Change the receiver's top-left corner's coordinates to aPoint

topRight Answer the receiver's top-right corner

topRight: aPoint
Change the receiver's top-right corner to aPoint

width Answer the receiver's width

width: aNumber
Change the receiver's width to aNumber

6.120.3 Rectangle: copying

copy Return a deep copy of the receiver for safety.

6.120.4 Rectangle: printing

printOn: aStream
Print a representation of the receiver on aStream

storeOn: aStream
Store Smalltalk code compiling to the receiver on aStream

6.120.5 Rectangle: rectangle functions

amountToTranslateWithin: aRectangle
Answer a Point so that if aRectangle is translated by that point, its origin lies within the receiver's.

area Answer the receiver's area. The area is the width times the height, so it is possible for it to be negative if the rectangle is not normalized.

areasOutside: aRectangle
Answer a collection of rectangles containing the parts of the receiver outside of aRectangle. For all points in the receiver, but outside aRectangle, exactly one rectangle in the collection will contain that point.

expandBy: delta
Answer a new rectangle that is the receiver expanded by aValue: if aValue is a rectangle, calculate origin=origin-aValue origin, corner=corner+aValue corner; else calculate origin=origin-aValue, corner=corner+aValue.

insetBy: delta
Answer a new rectangle that is the receiver inset by aValue: if aValue is a rectangle, calculate origin=origin+aValue origin, corner=corner-aValue corner; else calculate origin=origin+aValue, corner=corner-aValue.

insetOriginBy: originDelta corner: cornerDelta
Answer a new rectangle that is the receiver inset so that origin=origin+originDelta, corner=corner-cornerDelta. The deltas can be points or numbers

intersect: aRectangle
Returns the rectangle (if any) created by the overlap of rectangles A and B.

merge: aRectangle
Answer a new rectangle which is the smallest rectangle containing both the receiver and aRectangle.

translatedToBeWithin: aRectangle
Answer a copy of the receiver that does not extend beyond aRectangle.

6.120.6 Rectangle: testing

= aRectangle
Answer whether the receiver is equal to aRectangle

contains: aRectangle
Answer true if the receiver contains (see containsPoint:) both aRectangle's origin and aRectangle's corner

containsPoint: aPoint
Answer true if aPoint is equal to, or below and to the right of, the receiver's origin; and aPoint is above and to the left of the receiver's corner

hash Answer an hash value for the receiver

intersects: aRectangle
Answer true if the receiver intersect aRectangle, i.e. if it contains (see containsPoint:) any of aRectangle corners or if aRectangle contains the receiver

6.120.7 Rectangle: transforming

moveBy: aPoint
Change the receiver so that the origin and corner are shifted by aPoint

moveTo: aPoint
Change the receiver so that the origin moves to aPoint and the size remains unchanged

scaleBy: scale
Answer a copy of the receiver in which the origin and corner are multiplied by scale

translateBy: factor
Answer a copy of the receiver in which the origin and corner are shifted by aPoint

6.120.8 Rectangle: truncation and round off

rounded Answer a copy of the receiver with the coordinates rounded to the nearest integers

6.121 RootNamespace

Defined in namespace Smalltalk
Category: Language-Implementation
I am a special form of dictionary. I provide special ways to access my keys, which typically begin with an uppercase letter. Classes hold on an instance of me; it is called their 'environment').

My keys are (expected to be) symbols, so I use == to match searched keys to those in the dictionary – this is done expecting that it brings a bit more speed.

6.121.1 RootNamespace class: instance creation

new Disabled - use #new to create instances

new: spaceName
Create a new root namespace with the given name, and add to Smalltalk a key that references it.

primNew: parent name: spaceName
Private - Create a new namespace with the given name and parent, and add to the parent a key that references it.

6.121.2 RootNamespace: accessing

allAssociations
Answer a Dictionary with all of the associations in the receiver and each of its superspaces (duplicate keys are associated to the associations that are deeper in the namespace hierarchy)

allBehaviorsDo: aBlock
Evaluate aBlock once for each class and metaclass in the namespace.

allClassesDo: aBlock
Evaluate aBlock once for each class in the namespace.

allClassObjectsDo: aBlock
Evaluate aBlock once for each class and metaclass in the namespace.

allMetaclassesDo: aBlock
Evaluate aBlock once for each metaclass in the namespace.

classAt: aKey
Answer the value corrisponding to aKey if it is a class. Fail if either aKey is not found or it is associated to something different from a class.

classAt: aKey ifAbsent: aBlock
Answer the value corrisponding to aKey if it is a class. Evaluate aBlock and answer its result if either aKey is not found or it is associated to something different from a class.

define: aSymbol
Define aSymbol as equal to nil inside the receiver. Fail if such a variable already exists (use #at:put: if you don't want to fail)

doesNotUnderstand: aMessage
Try to map unary selectors to read accesses to the Namespace, and one-argument keyword selectors to write accesses. Note that: a) this works only if the selector has an uppercase first letter; and b) 'aNamespace Variable: value' is the same as 'aNamespace set: #Variable to: value', not the same as 'aNamespace at: #Variable put: value' — the latter always refers to the current namespace, while the former won't define a new variable, instead searching in superspaces (and raising an error if the variable cannot be found).

import: aSymbol from: aNamespace
Add to the receiver the symbol aSymbol, associated to the same value as in aNamespace. Fail if aNamespace does not contain the given key.

6.121.3 RootNamespace: basic & copying

= arg Answer whether the receiver is equal to arg. The equality test is by default the same as that for equal objects. = must not fail; answer false if the receiver cannot be compared to arg

identityHash
Answer an hash value for the receiver. This is the same as the object's -#identityHash.

6.121.4 RootNamespace: copying

copy Answer the receiver.

deepCopy Answer the receiver.

shallowCopy
Answer the receiver.

6.121.5 RootNamespace: forward declarations

at: key put: value
Store value as associated to the given key. If any, recycle Associations temporarily stored by the compiler inside the 'Undeclared' dictionary.

6.121.6 RootNamespace: namespace hierarchy

addSubspace: aSymbol
Add aNamespace to the set of the receiver's subspaces

allSubassociationsDo: aBlock
Invokes aBlock once for every association in each of the receiver's subspaces.

allSubspaces
Answer the direct and indirect subspaces of the receiver in a Set

allSubspacesDo: aBlock
Invokes aBlock for all subspaces, both direct and indirect.

allSuperspaces
Answer all the receiver's superspaces in a collection

allSuperspacesDo: aBlock
Evaluate aBlock once for each of the receiver's superspaces

includesClassNamed: aString
Answer whether the receiver or any of its superspaces include the given class – note that this method (unlike #includesKey:) does not require aString to be interned and (unlike #includesGlobalNamed:) only returns true if the global is a class object.

includesGlobalNamed: aString
Answer whether the receiver or any of its superspaces include the given key – note that this method (unlike #includesKey:) does not require aString to be interned but (unlike #includesClassNamed:) returns true even if the global is not a class object.

inheritsFrom: aNamespace
Answer whether aNamespace is one of the receiver's direct and indirect superspaces

selectSubspaces: aBlock
Return a Set of subspaces of the receiver satisfying aBlock.

selectSuperspaces: aBlock
Return a Set of superspaces of the receiver satisfying aBlock.

siblings Answer all the other root namespaces

siblingsDo: aBlock
Evaluate aBlock once for each of the other root namespaces, passing the namespace as a parameter.

subspaces Answer the receiver's direct subspaces

subspacesDo: aBlock
Invokes aBlock for all direct subspaces.

superspace
Send #at:ifAbsent: to super because our implementation of #at:ifAbsent: sends this message (chicken and egg!)

superspace: aNamespace
Set the superspace of the receiver to be 'aNamespace'. Also adds the receiver as a subspace of it.

withAllSubspaces
Answer a Set containing the receiver together with its direct and indirect subspaces

withAllSubspacesDo: aBlock
Invokes aBlock for the receiver and all subclasses, both direct and indirect.

withAllSuperspaces
Answer the receiver and all of its superspaces in a collection

withAllSuperspacesDo: aBlock
Invokes aBlock for the receiver and all superspaces, both direct and indirect.

6.121.7 RootNamespace: overrides for superspaces

definedKeys
Answer a kind of Set containing the keys of the receiver

definesKey: key
Answer whether the receiver defines the given key. 'Defines' means that the receiver's superspaces, if any, are not considered.

hereAt: key
Return the value associated to the variable named as specified by 'key' *in this namespace*. If the key is not found search will *not* be brought on in superspaces and the method will fail.

hereAt: key ifAbsent: aBlock
Return the value associated to the variable named as specified by 'key' *in this namespace*. If the key is not found search will *not* be brought on in superspaces and aBlock will be immediately evaluated.

inheritedKeys
Answer a Set of all the keys in the receiver and its superspaces

set: key to: newValue
Assign newValue to the variable named as specified by 'key'. This method won't define a new variable; instead if the key is not found it will search in superspaces and raising an error if the variable cannot be found in any of the superspaces. Answer newValue.

set: key to: newValue ifAbsent: aBlock
Assign newValue to the variable named as specified by 'key'. This method won't define a new variable; instead if the key is not found it will search in superspaces and evaluate aBlock if it is not found. Answer newValue.

values Answer a Bag containing the values of the receiver

6.121.8 RootNamespace: printing

defaultName
Private - Answer the name to be used if the receiver is not attached to an association in the superspace

name Answer the receiver's name

nameIn: aNamespace
Answer Smalltalk code compiling to the receiver when the current namespace is aNamespace

printOn: aStream
Print a representation of the receiver

storeOn: aStream
Store Smalltalk code compiling to the receiver

6.121.9 RootNamespace: testing

isNamespace
Answer 'true'.

isSmalltalk
Answer 'false'.

species Answer 'IdentityDictionary'.

6.122 RunArray

Defined in namespace Smalltalk
Category: Collection-Sequenceable
My instances are OrderedCollections that automatically apply Run Length Encoding compression to the things they store. Be careful when using me: I can provide great space savings, but my instances don't grant linear access time. RunArray's behavior currently is similar to that of OrderedCollection (you can add elements to RunArrays); maybe it should behave like an ArrayedCollection.

6.122.1 RunArray class: instance creation

new Answer an empty RunArray

new: aSize
Answer a RunArray with space for aSize runs

6.122.2 RunArray: accessing

at: anIndex
Answer the element at index anIndex

at: anIndex put: anObject
Replace the element at index anIndex with anObject and answer anObject

6.122.3 RunArray: adding

add: anObject afterIndex: anIndex
Add anObject after the element at index anIndex

addAll: aCollection afterIndex: anIndex
Add all the elements of aCollection after the one at index anIndex. If aCollection is unordered, its elements could be added in an order which is not the #do: order

addAllFirst: aCollection
Add all the elements of aCollection at the beginning of the receiver. If aCollection is unordered, its elements could be added in an order which is not the #do: order

addAllLast: aCollection
Add all the elements of aCollection at the end of the receiver. If aCol- lection is unordered, its elements could be added in an order which is not the #do: order

addFirst: anObject
Add anObject at the beginning of the receiver. Watch out: this operation can cause serious performance pitfalls

addLast: anObject
Add anObject at the end of the receiver

6.122.4 RunArray: basic

first Answer the first element in the receiver

last Answer the last element of the receiver

size Answer the number of elements in the receiver

6.122.5 RunArray: copying

deepCopy Answer a copy of the receiver containing copies of the receiver's elements (-#copy is used to obtain them)

shallowCopy
Answer a copy of the receiver. The elements are not copied

6.122.6 RunArray: enumerating

do: aBlock
Enumerate all the objects in the receiver, passing each one to aBlock

objectsAndRunLengthsDo: aBlock
Enumerate all the runs in the receiver, passing to aBlock two parameters for every run: the first is the repeated object, the second is the number of copies

6.122.7 RunArray: removing

removeAtIndex: anIndex
Remove the object at index anIndex from the receiver and answer the removed object

removeFirst
Remove the first object from the receiver and answer the removed object

removeLast
Remove the last object from the receiver and answer the removed object

6.122.8 RunArray: searching

indexOf: anObject startingAt: anIndex ifAbsent: aBlock
Answer the index of the first copy of anObject in the receiver, starting the search at the element at index anIndex. If no equal object is found, answer the result of evaluating aBlock

6.122.9 RunArray: testing

= anObject
Answer true if the receiver is equal to anObject

hash Answer an hash value for the receiver

6.123 ScaledDecimal

Defined in namespace Smalltalk
Category: Kernel-Numbers
ScaledDecimal provides a numeric representation of fixed point decimal numbers able to accurately represent decimal fractions. It supports unbounded precision, with no limit to the number of digits before and after the decimal point.

6.123.1 ScaledDecimal class: constants

initialize Initialize the receiver's class variables

6.123.2 ScaledDecimal class: instance creation

newFromNumber: aNumber scale: scale
Answer a new instance of ScaledDecimal, representing a decimal fraction with a decimal representation considered valid up to the scale-th digit.

6.123.3 ScaledDecimal: arithmetic

*** aNumber**
Multiply two numbers and answer the result.

+ aNumber
Sum two numbers and answer the result.

- aNumber
Subtract aNumber from the receiver and answer the result.

/ aNumber
Divide two numbers and answer the result.

// aNumber
Answer the integer quotient after dividing the receiver by aNumber with truncation towards negative infinity.

\\ aNumber
Answer the remainder after integer division the receiver by aNumber with truncation towards negative infinity.

6.123.4 ScaledDecimal: coercion

asFloat Answer the receiver, converted to a Float

asFraction Answer the receiver, converted to a Fraction

coerce: aNumber
Answer aNumber, converted to a ScaledDecimal with the same scale as the receiver.

fractionPart
Answer the fractional part of the receiver.

generality Return the receiver's generality

integerPart
Answer the fractional part of the receiver.

truncated Answer the receiver, converted to an Integer and truncated towards -infinity.

6.123.5 ScaledDecimal: comparing

< aNumber
Answer whether the receiver is less than arg.

<= aNumber
Answer whether the receiver is less than or equal to arg.

= arg Answer whether the receiver is equal to arg.

> aNumber
Answer whether the receiver is greater than arg.

>= aNumber
Answer whether the receiver is greater than or equal to arg.

hash Answer an hash value for the receiver.

~= arg Answer whether the receiver is not equal arg.

6.123.6 ScaledDecimal: constants

one Answer the receiver's representation of one.

zero Answer the receiver's representation of zero.

6.123.7 ScaledDecimal: printing

displayOn: aStream
Print a representation of the receiver on aStream, intended to be directed to a user. In this particular case, the 'scale' part of the #printString is not emitted.

printOn: aStream
Print a representation of the receiver on aStream.

6.123.8 ScaledDecimal: storing

storeOn: aStream
Print Smalltalk code that compiles to the receiver on aStream.

6.124 Semaphore

Defined in namespace Smalltalk
Category: Language-Processes
My instances represent counting semaphores. I provide methods for signalling the semaphore's availability, and methods for waiting for its availability. I also provide some methods for implementing critical sections. I currently do not (because the underlying system does not) support asynchronous signals, such as might be generated by C signals.

6.124.1 Semaphore class: instance creation

forMutualExclusion
Answer a new semaphore with a signal on it. These semaphores are a useful shortcut when you use semaphores as critical sections.

new Answer a new semaphore

6.124.2 Semaphore: builtins

signal Signal the receiver, resuming a waiting process' if there is one

wait Wait for the receiver to be signalled, suspending the executing process if it is not yet

6.124.3 Semaphore: mutual exclusion

critical: aBlock
Wait for the receiver to be free, execute aBlock and signal the receiver again. Return the result of evaluating aBlock. aBlock MUST NOT CONTAIN A RETURN!!!

6.125 SequenceableCollection

Defined in namespace Smalltalk
Category: Collections-Sequenceable

My instances represent collections of objects that are ordered. I provide some access and manipulation methods.

6.125.1 SequenceableCollection class: instance creation

streamContents: aBlock

Create a ReadWriteStream on an empty instance of the receiver; pass the stream to aBlock, then retrieve its contents and answer them.

6.125.2 SequenceableCollection: basic

after: oldObject

Return the element after oldObject. Error if oldObject not found or if no following object is available

atAll: aCollection put: anObject

Put anObject at every index contained in aCollection

atAllPut: anObject

Put anObject at every index in the receiver

before: oldObject

Return the element before oldObject. Error if oldObject not found or if no preceding object is available

first

Answer the first item in the receiver

identityIndexOf: anElement

Answer the index of the first occurrence of an object identical to anElement in the receiver. Answer 0 if no item is found

identityIndexOf: anElement ifAbsent: exceptionBlock

Answer the index of the first occurrence of an object identical to anElement in the receiver. Invoke exceptionBlock and answer its result if no item is found

identityIndexOf: anElement startingAt: anIndex

Answer the first index > anIndex which contains an object identical to anElement. Answer 0 if no item is found

identityIndexOf: anObject startingAt: anIndex ifAbsent: exceptionBlock

Answer the first index > anIndex which contains an object exactly identical to anObject. Invoke exceptionBlock and answer its result if no item is found

indexOf: anElement

Answer the index of the first occurrence of anElement in the receiver. Answer 0 if no item is found

indexOf: anElement ifAbsent: exceptionBlock
Answer the index of the first occurrence of anElement in the receiver. Invoke exceptionBlock and answer its result if no item is found

indexOf: anElement startingAt: anIndex
Answer the first index > anIndex which contains anElement. Answer 0 if no item is found

indexOf: anElement startingAt: anIndex ifAbsent: exceptionBlock
Answer the first index > anIndex which contains anElement. Invoke exceptionBlock and answer its result if no item is found

indexOfSubCollection: aSubCollection
Answer the first index > anIndex at which starts a sequence of items matching aSubCollection. Answer 0 if no such sequence is found.

indexOfSubCollection: aSubCollection ifAbsent: exceptionBlock
Answer the first index > anIndex at which starts a sequence of items matching aSubCollection. Answer 0 if no such sequence is found.

indexOfSubCollection: aSubCollection startingAt: anIndex
Answer the first index > anIndex at which starts a sequence of items matching aSubCollection. Answer 0 if no such sequence is found.

indexOfSubCollection: aSubCollection startingAt: anIndex ifAbsent: exceptionBlock
Answer the first index > anIndex at which starts a sequence of items matching aSubCollection. Invoke exceptionBlock and answer its result if no such sequence is found

last Answer the last item in the receiver

6.125.3 SequenceableCollection: copying SequenceableCollections

, aSequenceableCollection
Append aSequenceableCollection at the end of the receiver (using #add:), and answer a new collection

copyFrom: start
Answer a new collection containing all the items in the receiver from the start-th.

copyFrom: start to: stop
Answer a new collection containing all the items in the receiver from the start-th and to the stop-th

copyReplaceAll: oldSubCollection with: newSubCollection
Answer a new collection in which all the sequences matching oldSubCollection are replaced with newSubCollection

copyReplaceFrom: start to: stop with: replacementCollection
Answer a new collection of the same class as the receiver that contains the same elements as the receiver, in the same order, except for elements from index 'start' to index 'stop'. If start < stop, these are replaced by the contents of the

replacementCollection. Instead, If start = (stop + 1), like in 'copyReplaceFrom: 4 to: 3 with: anArray', then every element of the receiver will be present in the answered copy; the operation will be an append if stop is equal to the size of the receiver or, if it is not, an insert before index 'start'.

copyReplaceFrom: start to: stop withObject: anObject
Answer a new collection of the same class as the receiver that contains the same elements as the receiver, in the same order, except for elements from index 'start' to index 'stop'. If start < stop, these are replaced by the single element anObject. Instead, If start = (stop + 1), then every element of the receiver will be present in the answered copy; the operation will be an append if stop is equal to the size of the receiver or, if it is not, an insert before index 'start'.

6.125.4 SequenceableCollection: enumerating

anyOne Answer an unspecified element of the collection. Example usage: ^coll inject: coll anyOne into: [:max :each | max max: each] to be used when you don't have a valid lowest-possible-value (which happens in common cases too, such as with arbitrary numbers

do: aBlock
Evaluate aBlock for all the elements in the sequenceable collection

do: aBlock separatedBy: sepBlock
Evaluate aBlock for all the elements in the sequenceable collection. Between each element, evaluate sepBlock without parameters.

doWithIndex: aBlock
Evaluate aBlock for all the elements in the sequenceable collection, passing the index of each element as the second parameter. This method is mantained for backwards compatibility and is not mandated by the ANSI standard; use #keysAndValuesDo:

findFirst: aBlock
Returns the index of the first element of the sequenceable collection for which aBlock returns true, or 0 if none

findLast: aBlock
Returns the index of the last element of the sequenceable collection for which aBlock returns true, or 0 if none does

from: startIndex to: stopIndex do: aBlock
Evaluate aBlock for all the elements in the sequenceable collection whose indices are in the range index to stopIndex

from: startIndex to: stopIndex doWithIndex: aBlock
Evaluate aBlock for all the elements in the sequenceable collection whose indices are in the range index to stopIndex, passing the index of each element as the second parameter. This method is mantained for backwards compatibility and is not mandated by the ANSI standard; use #from:to:keysAndValuesDo:

from: startIndex to: stopIndex keysAndValuesDo: aBlock
Evaluate aBlock for all the elements in the sequenceable collection whose indices are in the range index to stopIndex, passing the index of each element as the first parameter and the element as the second.

keysAndValuesDo: aBlock
Evaluate aBlock for all the elements in the sequenceable collection, passing the index of each element as the first parameter and the element as the second.

readStream
Answer a ReadStream streaming on the receiver

readWriteStream
Answer a ReadWriteStream which streams on the receiver

reverse Answer the receivers' contents in reverse order

reverseDo: aBlock
Evaluate aBlock for all elements in the sequenceable collection, from the last to the first.

with: aSequenceableCollection collect: aBlock
Evaluate aBlock for each pair of elements took respectively from the re- ceiver and from aSequenceableCollection; answer a collection of the same kind of the receiver, made with the block's return values. Fail if the receiver has not the same size as aSequenceableCollection.

with: aSequenceableCollection do: aBlock
Evaluate aBlock for each pair of elements took respectively from the re- ceiver and from aSequenceableCollection. Fail if the receiver has not the same size as aSequenceableCollection.

writeStream
Answer a WriteStream streaming on the receiver

6.125.5 SequenceableCollection: replacing items

replaceAll: anObject with: anotherObject
In the receiver, replace every occurrence of anObject with anotherObject.

replaceFrom: start to: stop with: replacementCollection
Replace the items from start to stop with replacementCollection's items from 1 to stop-start+1 (in unexpected order if the collection is not sequenceable).

replaceFrom: start to: stop with: replacementCollection startingAt: repStart
Replace the items from start to stop with replacementCollection's items from repStart to repStart+stop-start

replaceFrom: anIndex to: stopIndex withObject: replacementObject
Replace every item from start to stop with replacementObject.

6.125.6 SequenceableCollection: testing

= aCollection
Answer whether the receiver's items match those in aCollection

hash Answer an hash value for the receiver

inspect Print all the instance variables and context of the receiver on the Transcript

6.126 Set

Defined in namespace Smalltalk
Category: Collections-Unordered
I am the typical set object; I also known how to do arithmetic on my instances.

6.126.1 Set: arithmetic

& aSet Compute the set intersection of the receiver and aSet.

+ aSet Compute the set union of the receiver and aSet.

- aSet Compute the set difference of the receiver and aSet.

6.126.2 Set: awful ST-80 compatibility hacks

findObjectIndex: object
Tries to see if anObject exists as an indexed variable. As soon as nil or anObject is found, the index of that slot is answered

6.126.3 Set: comparing

< aSet Answer whether the receiver is a strict subset of aSet

<= aSet Answer whether the receiver is a subset of aSet

> aSet Answer whether the receiver is a strict superset of aSet

>= aSet Answer whether the receiver is a superset of aSet

6.127 SharedQueue

Defined in namespace Smalltalk
Category: Language-Processes
My instances provide a guaranteed safe mechanism to allow for communication between processes. All access to the underlying data structures is controlled with critical sections so that things proceed smoothly.

6.127.1 SharedQueue class: instance creation

new Create a new instance of the receiver

sortBlock: sortBlock
Create a new instance of the receiver which implements a priority queue with the given sort block

6.127.2 SharedQueue: accessing

next Wait for an object to be on the queue, then remove it and answer it

nextPut: value
Put value on the queue and answer it

peek Wait for an object to be on the queue if necessary, then answer the same object that #next would answer without removing it.

6.128 Signal

Defined in namespace Smalltalk
Category: Language-Exceptions
My instances describe an exception that has happened, and are passed to exception handlers. Apart from containing information on the generated exception and its arguments, they contain methods that allow you to resume execution, leave the #on:do:... snippet, and pass the exception to an handler with a lower priority.

6.128.1 Signal: accessing

argument Answer the first argument of the receiver

argumentCount
Answer how many arguments the receiver has

arguments Answer the arguments of the receiver

basicMessageText
Answer an exception's message text. Do not override this method.

description
Answer the description of the raised exception

exception Answer the CoreException that was raised

messageText
Answer an exception's message text.

messageText: aString
Set an exception's message text.

tag Answer an exception's tag value. If not specified, it is the same as the message text.

tag: anObject
Set an exception's tag value. If nil, the tag value will be the same as the message text.

6.128.2 Signal: exception handling

defaultAction
Execute the default handler for the raised exception

isNested Answer whether the current exception handler is within the scope of another handler for the same exception.

isResumable
Answer whether the exception that instantiated the receiver is resumable.

outer Raise the exception that instantiated the receiver, passing the same parameters. If the receiver is resumable and the evaluated exception action resumes then the result returned from #outer will be the resumption value of the evaluated exception action. If the receiver is not resumable or if the exception action does not resume then this message will not return, and #outer will be equivalent to #pass.

pass Yield control to the enclosing exception action for the receiver. Similar to #outer, but control does not return to the currently active exception handler.

resignalAs: replacementException
Reinstate all handlers and execute the handler for 'replacementException'; control does not return to the currently active exception handler. The new Signal object that is created has the same arguments as the receiver (this might or not be correct – if it isn't you can use an idiom such as 'sig retryUsing: [replacementException signal])

resume If the exception is resumable, resume the execution of the block that raised the exception; the method that was used to signal the exception will answer the receiver. Use this method IF AND ONLY IF you know who caused the exception and if it is possible to resume it in that particular case

resume: anObject
If the exception is resumable, resume the execution of the block that raised the exception; the method that was used to signal the exception will answer anObject. Use this method IF AND ONLY IF you know who caused the exception and if it is possible to resume it in that particular case

retry Re-execute the receiver of the #on:do: message. All handlers are reinstated: watch out, this can easily cause an infinite loop.

retryUsing: aBlock
Execute aBlock reinstating all handlers, and return its result from the #signal method.

return Exit the #on:do: snippet, answering anObject to its caller

return: anObject
Exit the #on:do: snippet, answering anObject to its caller

6.129 SingletonProxy

Defined in namespace Smalltalk
Category: Streams-Files

6.129.1 SingletonProxy class: accessing

acceptUsageForClass: aClass
The receiver was asked to be used as a proxy for the class aClass. The registration is fine if the class is actually a singleton.

6.129.2 SingletonProxy class: instance creation

on: anObject
Answer a proxy to be used to save anObject. The proxy stores the class and restores the object by looking into a dictionary of class -> singleton objects.

6.129.3 SingletonProxy: saving and restoring

object Reconstruct the object stored in the proxy and answer it; the binaryRepresentationObject is sent the #reconstructOriginalObject message, and the resulting object is sent the #postLoad message.

6.130 SmallInteger

Defined in namespace Smalltalk
Category: Language-Data types

6.130.1 SmallInteger: built ins

*** arg** Multiply the receiver and arg and answer another Number

+ arg Sum the receiver and arg and answer another Number

- arg Subtract arg from the receiver and answer another Number

/ arg Divide the receiver by arg and answer another Integer or Fraction

// arg Dividing receiver by arg (with truncation towards -infinity) and answer the result

< arg Answer whether the receiver is less than arg

<= arg Answer whether the receiver is less than or equal to arg

= arg Answer whether the receiver is equal to arg

== arg Answer whether the receiver is the same object as arg

> arg Answer whether the receiver is greater than arg

>= arg Answer whether the receiver is greater than or equal to arg

\\ arg Calculate the remainder of dividing receiver by arg (with truncation towards -infinity) and answer it

asFloat Convert the receiver to a Float, answer the result

asObject Answer the object whose index is in the receiver, fail if no object found at that index

asObjectNoFail
Answer the object whose index is in the receiver, or nil if no object is found at that index

bitAnd: arg
Do a bitwise AND between the receiver and arg, answer the result

bitOr: arg Do a bitwise OR between the receiver and arg, answer the result

bitShift: arg
Shift the receiver by arg places to the left if arg > 0, by arg places to the right if arg < 0, answer another Number

bitXor: arg
Do a bitwise XOR between the receiver and arg, answer the result

quo: arg Dividing receiver by arg (with truncation towards zero) and answer the result

~= arg Answer whether the receiver is not equal to arg

~~ arg Answer whether the receiver is not the same object as arg

6.130.2 SmallInteger: builtins

at: anIndex
Answer the index-th indexed instance variable of the receiver. This method always fails.

at: anIndex put: value
Store value in the index-th indexed instance variable of the receiver This method always fails.

basicAt: anIndex
Answer the index-th indexed instance variable of the receiver. This method always fails.

basicAt: anIndex put: value
Store value in the index-th indexed instance variable of the receiver This method always fails.

6.131 SortedCollection

Defined in namespace Smalltalk
Category: Collections-Sequenceable

I am a collection of objects, stored and accessed according to some sorting criteria. I store things using heap sort and quick sort. My instances have a comparison block associated with them; this block takes two arguments and is a predicate which returns true if the first argument should be sorted earlier than the second. The default block is [:a :b | a <= b], but I will accept any block that conforms to the above criteria – actually any object which responds to #value:value:.

6.131.1 SortedCollection class: hacking

defaultSortBlock
Answer a default sort block for the receiver.

6.131.2 SortedCollection class: instance creation

new Answer a new collection with a default size and sort block

new: aSize
Answer a new collection with a default sort block and the given size

sortBlock: aSortBlock
Answer a new collection with a default size and the given sort block

6.131.3 SortedCollection: basic

last Answer the last item of the receiver

removeLast
Remove an object from the end of the receiver. Fail if the receiver is empty

sortBlock Answer the receiver's sort criteria

sortBlock: aSortBlock
Change the sort criteria for a sorted collection, resort the elements of the collection, and return it.

6.131.4 SortedCollection: copying

copyEmpty: newSize
Answer an empty copy of the receiver, with the same sort block as the receiver

6.131.5 SortedCollection: disabled

add: anObject afterIndex: i
This method should not be called for instances of this class.

addAll: aCollection afterIndex: i
This method should not be called for instances of this class.

addAllFirst: aCollection
This method should not be called for instances of this class.

addAllLast: aCollection
This method should not be called for instances of this class.

addFirst: anObject
This method should not be called for instances of this class.

addLast: anObject
This method should not be called for instances of this class.

at: index put: anObject
This method should not be called for instances of this class.

6.131.6 SortedCollection: enumerating

beConsistent
Prepare the receiver to be walked through with #do: or another enumeration method.

6.131.7 SortedCollection: saving and loading

postLoad Restore the default sortBlock if it is nil

preStore Store the default sortBlock as nil

6.131.8 SortedCollection: searching

includes: anObject
Private - Answer whether the receiver includes an item which is equal to anObject

indexOf: anObject startingAt: index ifAbsent: aBlock
Answer the first index > anIndex which contains anElement. Invoke exceptionBlock and answer its result if no item is found

occurrencesOf: anObject
Answer how many occurrences of anObject can be found in the receiver

6.132 Stream

Defined in namespace Smalltalk
Category: Streams

I am an abstract class that provides interruptable sequential access to objects. I can return successive objects from a source, or accept successive objects and store them sequentially on a sink. I provide some simple iteration over the contents of one of my instances, and provide for writing collections sequentially.

6.132.1 Stream: accessing-reading

contents Answer the whole contents of the receiver, from the next object to the last

next Return the next object in the receiver

next: anInteger
Return the next anInteger objects in the receiver

nextAvailable: anInteger
Return up to anInteger objects in the receiver, stopping if the end of the stream is reached

nextMatchFor: anObject
Answer whether the next object is equal to anObject. Even if it does not, anObject is lost

splitAt: anObject
Answer an OrderedCollection of parts of the receiver. A new (possibly empty) part starts at the start of the receiver, or after every occurrence of an object which is equal to anObject (as compared by #=).

6.132.2 Stream: accessing-writing

next: anInteger put: anObject
Write anInteger copies of anObject to the receiver

nextPut: anObject
Write anObject to the receiver

nextPutAll: aCollection
Write all the objects in aCollection to the receiver

6.132.3 Stream: basic

species Answer 'Array'.

6.132.4 Stream: character writing

cr Store a cr on the receiver

crTab Store a cr and a tab on the receiver

nl Store a new line on the receiver

nlTab Store a new line and a tab on the receiver

space Store a space on the receiver

space: n Store n spaces on the receiver

tab Store a tab on the receiver

tab: n Store n tabs on the receiver

6.132.5 Stream: enumerating

do: aBlock
Evaluate aBlock once for every object in the receiver

6.132.6 Stream: filing out

fileOut: aClass
File out aClass on the receiver. If aClass is not a metaclass, file out class and instance methods; if aClass is a metaclass, file out only the class methods

6.132.7 Stream: PositionableStream methods

nextLine Returns a collection of the same type that the stream accesses, up to but not including the object anObject. Returns the entire rest of the stream's contents if anObject is not present.

skip: anInteger
Move the position forwards by anInteger places

skipTo: anObject
Move the current position to after the next occurrence of anObject and return true if anObject was found. If anObject doesn't exist, the pointer is atEnd, and false is returned.

skipToAll: aCollection
If there is a sequence of objects remaining in the stream that is equal to the sequence in aCollection, set the stream position just past that sequence and answer true. Else, set the stream position to its end and answer false.

upTo: anObject
Returns a collection of the same type that the stream accesses, up to but not including the object anObject. Returns the entire rest of the stream's contents if anObject is not present.

upToAll: aCollection
: If there is a sequence of objects remaining in the stream that is equal to the sequence in aCollection, set the stream position just past that sequence and answer the elements up to, but not including, the sequence. Else, set the stream position to its end and answer all the remaining elements.

upToEnd Answer every item in the collection on which the receiver is streaming, from the next one to the last

6.132.8 Stream: printing

<< anObject
: This method is a short-cut for #display:; it prints anObject on the receiver by sending displayOn: to anObject. This method is provided so that you can use cascading and obtain better-looking code

display: anObject
: Print anObject on the receiver by sending displayOn: to anObject. This method is provided so that you can use cascading and obtain better-looking code

print: anObject
: Print anObject on the receiver by sending printOn: to anObject. This method is provided so that you can use cascading and obtain better-looking code

6.132.9 Stream: providing consistent protocols

close Do nothing. This is provided for consistency with file streams

flush Do nothing. This is provided for consistency with file streams

6.132.10 Stream: storing

store: anObject
: Print Smalltalk code compiling to anObject on the receiver, by sending storeOn: to anObject. This method is provided so that you can use cascading and obtain better-looking code

6.132.11 Stream: testing

atEnd Answer whether the stream has got to an end

6.133 String

Defined in namespace Smalltalk
Category: Language-Data types
My instances represent ASCII string data types. Being a very common case, they are particularly optimized.

6.133.1 String class: basic

, aString Answer a new instance of an ArrayedCollection containing all the elements in the receiver, followed by all the elements in aSequenceableCollection

6.133.2 String class: instance creation

fromCData: aCObject size: anInteger
Answer a String containing anInteger bytes starting at the location pointed to by aCObject

6.133.3 String: built ins

asCData: aCType
Convert the receiver to a CObject with the given type

at: index Answer the index-th character of the receiver.

at: index put: value
Change the index-th character of the receiver.

basicAt: index
Answer the index-th character of the receiver. This method must not be overridden; override at: instead. String overrides it so that it looks like it contains character objects even though it contains bytes

basicAt: index put: value
Change the index-th character of the receiver. This method must not be overridden; override at: instead. String overrides it so that it looks like it contains character objects even though it contains bytes

hash Answer an hash value for the receiver

primReplaceFrom: start to: stop with: replacementString
startingAt: replaceStart Private - Replace the characters from start to stop with new characters contained in replacementString (which, actually, can be any variable byte class, starting at the replaceStart location of replacementString

replaceFrom: start to: stop with: aString startingAt: replaceStart
Replace the characters from start to stop with new characters whose ASCII codes are contained in aString, starting at the replaceStart location of aString

replaceFrom: start to: stop withByteArray: byteArray startingAt: replaceStart
Replace the characters from start to stop with new characters whose ASCII codes are contained in byteArray, starting at the replaceStart location of byteArray

size Answer the size of the receiver

6.133.4 String: converting

asByteArray
Return the receiver, converted to a ByteArray of ASCII values

asString But I already am a String! Really!

asSymbol Returns the symbol corresponding to the receiver

6.133.5 String: storing

storeOn: aStream
Print Smalltalk code compiling to the receiver on aStream

6.133.6 String: testing functionality

isString Answer 'true'.

6.133.7 String: useful functionality

linesDo: aBlock
Send 'aBlock' a substring of the receiver for each newline delimited line in the receiver

6.134 Symbol

Defined in namespace Smalltalk
Category: Language-Implementation
My instances are unique throughout the Smalltalk system. My instances behave for the most part like strings, except that they print differently, and I guarantee that any two instances that have the same printed representation are in fact the same instance.

6.134.1 Symbol class: built ins

intern: aString
Private - Same as 'aString asSymbol'

6.134.2 Symbol class: instance creation

internCharacter: aCharacter
Answer the one-character symbol associated to the given character.

new This method should not be called for instances of this class.

new: size This method should not be called for instances of this class.

with: element1
Answer a collection whose only element is element1

with: element1 with: element2
Answer a collection whose only elements are the parameters in the order they were passed

with: element1 with: element2 with: element3
Answer a collection whose only elements are the parameters in the order they were passed

with: element1 with: element2 with: element3 with: element4
Answer a collection whose only elements are the parameters in the order they were passed

with: element1 with: element2 with: element3 with: element4 with: element5
Answer a collection whose only elements are the parameters in the order they were passed

6.134.3 Symbol class: symbol table

hasInterned: aString ifTrue: aBlock
If aString has not been interned yet, answer false. Else, pass the interned version to aBlock and answer true. Note that this works because String>>#hash calculates the same hash value used by the VM when interning strings into the SymbolTable. Changing one of the hashing methods without changing the other will break this method.

isSymbolString: aString
Answer whether aString has already been interned. Note that this works because String>>#hash calculates the same hash value used by the VM when interning strings into the SymbolTable. Changing one of the hashing methods without changing the other will break this method.

rebuildTable
Rebuild the SymbolTable, thereby garbage-collecting unreferenced Symbols. While this process is done, preemption is disabled because it is not acceptable to leave the SymbolTable in a partially updated state. Note that this works because String>>#hash calculates the same hash value used by the VM when interning strings into the SymbolTable. Changing one of the hashing methods without changing the other will break this method.

6.134.4 Symbol: basic

deepCopy Returns a deep copy of the receiver. As Symbols are identity objects, we actually return the receiver itself.

numArgs Answer the number of arguments supported by the receiver, which is supposed to be a valid message name (#+, #not, #printOn:, #ifTrue:ifFalse:, etc.)

shallowCopy
Returns a deep copy of the receiver. As Symbols are identity objects, we actually return the receiver itself.

6.134.5 Symbol: built ins

= aSymbol
Answer whether the receiver and aSymbol are the same object

hash Answer an hash value for the receiver. Symbols are optimized for speed

6.134.6 Symbol: converting

asString Answer a String with the same characters as the receiver

asSymbol But we are already a Symbol, and furthermore, Symbols are identity objects! So answer the receiver.

6.134.7 Symbol: misc

species Answer 'String'.

6.134.8 Symbol: storing

displayOn: aStream
Print a represention of the receiver on aStream. For most objects this is simply its #printOn: representation, but for strings and characters, superfluous dollars or extra pairs of quotes are stripped.

displayString
Answer a String representing the receiver. For most objects this is simply its #printString, but for strings and characters, superfluous dollars or extra pair of quotes are stripped.

printOn: aStream
Print a represention of the receiver on aStream.

6.134.9 Symbol: testing

isSimpleSymbol
Answer whether the receiver must be represented in quoted-string (e.g. #'abc-def') form.

6.134.10 Symbol: testing functionality

isString Answer 'false'.

isSymbol Answer 'true'.

6.135 SymLink

Defined in namespace Smalltalk
Category: Language-Implementation

I am used to implement the Smalltalk symbol table. My instances are links that contain symbols, and the symbol table basically a hash table that points to chains of my instances.

6.135.1 SymLink class: instance creation

symbol: aSymbol nextLink: aSymLink

Answer a new SymLink, which refers to aSymbol and points to aSymLink as the next SymLink in the chain.

6.135.2 SymLink: accessing

symbol Answer the Symbol that the receiver refers to in the symbol table.

symbol: aSymbol

Set the Symbol that the receiver refers to in the symbol table.

6.135.3 SymLink: iteration

do: aBlock

Evaluate aBlock for each symbol in the list

6.135.4 SymLink: printing

printOn: aStream

Print a representation of the receiver on aStream.

6.136 SystemDictionary

Defined in namespace Smalltalk
Category: Language-Implementation

I am a special namespace. I only have one instance, called "Smalltalk", which is known to the Smalltalk interpreter. I define several methods that are "system" related, such as #quitPrimitive. My instance also helps keep track of dependencies between objects.

6.136.1 SystemDictionary: basic

halt Interrupt interpreter

hash Smalltalk usually contains a reference to itself, avoid infinite loops

6.136.2 SystemDictionary: builtins

byteCodeCounter
Answer the number of bytecodes executed by the VM

compact Force a full garbage collection. Sending this method to Smalltalk is deprecated; send it to ObjectMemory instead.

debug for GDB. Set breakpoint in debug() and invoke this primitive near where you want to stop

declarationTrace
Answer whether compiled bytecodes are printed on stdout

declarationTrace: aBoolean
Set whether compiled bytecodes are printed on stdout

executionTrace
Answer whether executed bytecodes are printed on stdout

executionTrace: aBoolean
Set whether executed bytecodes are printed on stdout

gcMessage Answer whether messages indicating that garbage collection is taking place are printed on stdout

gcMessage: aBoolean
Set whether messages indicating that garbage collection is taking place are printed on stdout

getTraceFlag: anIndex
Private - Returns a boolean value which is one of the interpreter's tracing flags

growThresholdPercent
Answer the percentage of the amount of memory used by the system grows which has to be full for the system to allocate more memory. Sending this method to Smalltalk is deprecated; send it to ObjectMemory instead.

growThresholdPercent: growPercent
Set the percentage of the amount of memory used by the system grows which has to be full for the system to allocate more memory. Sending this method to Smalltalk is deprecated; send it to ObjectMemory instead.

growTo: numBytes
Grow the amount of memory used by the system grows to numBytes. Sending this method to Smalltalk is deprecated; send it to ObjectMemory instead.

monitor: aBoolean
Start or stop profiling the VM's execution (if GNU Smalltalk was compiled with support for monitor(2), of course).

printStatistics
Print statistics about what the VM did since #resetStatistics was last called. Meaningful only if gst was made with 'make profile' or 'make profile_vm'. Sending this method to Smalltalk is deprecated; send it to ObjectMemory instead.

quitPrimitive
Quit the Smalltalk environment. Whether files are closed and other similar cleanup occurs depends on the platform. Sending this method to Smalltalk is deprecated; send #quit to ObjectMemory instead.

quitPrimitive: exitStatus
Quit the Smalltalk environment, passing the exitStatus integer to the OS. Whether files are closed and other similar cleanup occurs depends on the platform. Sending this method to Smalltalk is deprecated; send #quit to ObjectMemory instead.

resetStatistics
Reset the statistics about the VM which #printStatistics can print. Sending this method to Smalltalk is deprecated; send it to ObjectMemory instead.

setTraceFlag: anIndex to: aBoolean
Private - Sets the value of one of the interpreter's tracing flags (indicated by 'anIndex') to the value aBoolean.

snapshot: aString
Save an image on the aString file. Sending this method to Smalltalk is deprecated; send it to ObjectMemory instead.

spaceGrowRate
Answer the rate with which the amount of memory used by the system grows. Sending this method to Smalltalk is deprecated; send it to ObjectMemory instead.

spaceGrowRate: rate
Set the rate with which the amount of memory used by the system grows. Sending this method to Smalltalk is deprecated; send it to ObjectMemory instead.

verboseTrace
Answer whether execution tracing prints the object on the stack top

verboseTrace: aBoolean
Set whether execution tracing prints the object on the stack top

6.136.3 SystemDictionary: C functions

getArgc C call-out to getArgc. Do not modify!

getArgv: index
C call-out to getArgv. Do not modify!

getenv: aString
C call-out to getenv. Do not modify!

putenv: aString
C call-out to putenv. Do not modify!

system: aString
C call-out to system. Do not modify!

6.136.4 SystemDictionary: initialization

addInit: aBlock
Adds 'aBlock' to the array of blocks to be invoked after every start of the system. This mechanism is deprecated and will disappear in a future version; register your class as a dependent of ObjectMemory instead.

doInits Called after the system has loaded the image, this will invoke any init blocks that have been installed. This mechanism is deprecated; register your class as a dependent of ObjectMemory instead.

6.136.5 SystemDictionary: miscellaneous

arguments Return the command line arguments after the -a switch

backtrace Print a backtrace on the Transcript.

snapshot Save a snapshot on the image file that was loaded on startup. Sending this method to Smalltalk is deprecated; send it to ObjectMemory instead.

6.136.6 SystemDictionary: printing

defaultName
Answer ''Smalltalk".

name Answer the receiver's name

nameIn: aNamespace
Answer ''Smalltalk".

storeOn: aStream
Store Smalltalk code compiling to the receiver

6.136.7 SystemDictionary: special accessing

addFeature: aFeature
Add the aFeature feature to the Features set

dependenciesAt: anObject
Answer the dependants of anObject (or nil if there's no key for anObject in the Dependencies IdentityDictionary)

hasFeatures: features
Returns true if the feature or features in 'features' is one of the implementation dependent features present

removeFeature: aFeature
Remove the aFeature feature to the Features set

version Answer the current version of the GNU Smalltalk environment

6.137 SystemExceptions AlreadyDefined

Defined in namespace Smalltalk SystemExceptions
Category: Language-Exceptions
I am raised when one tries to define a symbol (class or pool variable) that is already defined.

6.137.1 SystemExceptions AlreadyDefined: accessing

description
Answer a description for the error

6.138 SystemExceptions ArgumentOutOfRange

Defined in namespace Smalltalk SystemExceptions
Category: Language-Exceptions
I am raised when one invokes a method with an argument outside of its valid range.

6.138.1 SystemExceptions ArgumentOutOfRange class: signaling

signalOn: value mustBeBetween: low and: high
Raise the exception. The given value was not between low and high.

6.138.2 SystemExceptions ArgumentOutOfRange: accessing

description
Answer a textual description of the exception.

high Answer the highest value that was permitted.

high: aMagnitude
Set the highest value that was permitted.

low Answer the lowest value that was permitted.

low: aMagnitude
Set the lowest value that was permitted.

6.139 SystemExceptions BadReturn

Defined in namespace Smalltalk SystemExceptions
Category: Language-Exceptions

I am raised when one tries to evaluate a method (via #perform:...) or a block but passes the wrong number of arguments.

6.139.1 SystemExceptions BadReturn: accessing

description

Answer a textual description of the exception.

6.140 SystemExceptions CInterfaceError

Defined in namespace Smalltalk SystemExceptions
Category: Language-Exceptions

I am raised when an error happens that is related to the C interface.

6.140.1 SystemExceptions CInterfaceError: accessing

description

Answer a textual description of the exception.

6.141 SystemExceptions EmptyCollection

Defined in namespace Smalltalk SystemExceptions
Category: Language-Exceptions

I am raised when one invokes a method on an empty collection.

6.141.1 SystemExceptions EmptyCollection: accessing

description

Answer a textual description of the exception.

6.142 SystemExceptions EndOfStream

Defined in namespace Smalltalk SystemExceptions
Category: Language-Exceptions

I am raised when a stream reaches its end

6.142.1 SystemExceptions EndOfStream class: signaling

signalOn: stream

Answer an exception reporting the parameter has reached its end.

6.142.2 SystemExceptions EndOfStream: accessing

description
Answer a textual description of the exception.

stream Answer the stream whose end was reached.

stream: anObject
Set the stream whose end was reached.

6.143 SystemExceptions FileError

Defined in namespace Smalltalk SystemExceptions
Category: Language-Exceptions
I am raised when an error happens that is related to the file system.

6.143.1 SystemExceptions FileError: accessing

description
Answer a textual description of the exception.

6.144 SystemExceptions IndexOutOfRange

Defined in namespace Smalltalk SystemExceptions
Category: Language-Exceptions
I am raised when one invokes am accessor method with an index outside of its valid range.

6.144.1 SystemExceptions IndexOutOfRange class: signaling

signalOn: aCollection withIndex: value
The given index was out of range in aCollection.

6.144.2 SystemExceptions IndexOutOfRange: accessing

collection Answer the collection that triggered the error

collection: anObject
Set the collection that triggered the error

description
Answer a textual description of the exception.

messageText
Answer an exception's message text.

6.145 SystemExceptions InvalidArgument

Defined in namespace Smalltalk SystemExceptions
Category: Language-Exceptions

I am raised when an argument is constrained to be an instance of a determinate class, and this constraint is not respected by the caller.

6.145.1 SystemExceptions InvalidArgument: accessing

messageText

Answer an exception's message text.

6.146 SystemExceptions InvalidSize

Defined in namespace Smalltalk SystemExceptions
Category: Language-Exceptions

I am raised when an argument has an invalid size.

6.146.1 SystemExceptions InvalidSize: accessing

description

Answer a textual description of the exception.

6.147 SystemExceptions InvalidValue

Defined in namespace Smalltalk SystemExceptions
Category: Language-Exceptions

I am raised when one invokes a method with an invalid argument.

6.147.1 SystemExceptions InvalidValue class: signaling

signalOn: value

Answer an exception reporting the parameter as invalid.

signalOn: value reason: reason

Answer an exception reporting 'value' as invalid, for the given reason.

6.147.2 SystemExceptions InvalidValue: accessing

description

Answer a textual description of the exception.

messageText

Answer an exception's message text.

value Answer the object that was found to be invalid.

value: anObject

Set the object that was found to be invalid.

6.148 SystemExceptions MustBeBoolean

Defined in namespace Smalltalk SystemExceptions
Category: Language-Exceptions
I am raised when one invokes a boolean method on a non-boolean.

6.149 SystemExceptions NoRunnableProcess

Defined in namespace Smalltalk SystemExceptions
Category: Language-Exceptions
I am raised when no runnable process can be found in the image.

6.149.1 SystemExceptions NoRunnableProcess: accessing

description
Answer a textual description of the exception.

6.150 SystemExceptions NotFound

Defined in namespace Smalltalk SystemExceptions
Category: Language-Exceptions
I am raised when something is searched without success.

6.150.1 SystemExceptions NotFound class: accessing

signalOn: value what: aString
Raise an exception; aString specifies what was not found (a key, an object, a class, and so on).

6.150.2 SystemExceptions NotFound: accessing

description
Answer a textual description of the exception.

6.151 SystemExceptions NotImplemented

Defined in namespace Smalltalk SystemExceptions
Category: Language-Exceptions
I am raised when a method is called that has not been implemented.

6.151.1 SystemExceptions NotImplemented: accessing

description
Answer a textual description of the exception.

6.152 SystemExceptions NotIndexable

Defined in namespace Smalltalk SystemExceptions
Category: Language-Exceptions
I am raised when an object is not indexable.

6.152.1 SystemExceptions NotIndexable: accessing

description
Answer a textual description of the exception.

6.153 SystemExceptions NotYetImplemented

Defined in namespace Smalltalk SystemExceptions
Category: Language-Exceptions
I am raised when a method is called that has not been implemented yet.

6.153.1 SystemExceptions NotYetImplemented: accessing

description
Answer a textual description of the exception.

6.154 SystemExceptions PrimitiveFailed

Defined in namespace Smalltalk SystemExceptions
Category: Language-Exceptions
I am raised when a primitive fails for some reason.

6.154.1 SystemExceptions PrimitiveFailed: accessing

description
Answer a textual description of the exception.

6.155 SystemExceptions ProcessTerminated

Defined in namespace Smalltalk SystemExceptions
Category: Language-Exceptions
I am raised when somebody tries to resume or interrupt a terminated process.

6.155.1 SystemExceptions ProcessTerminated: accessing

description
Answer a textual description of the exception.

6.156 SystemExceptions ReadOnlyObject

Defined in namespace Smalltalk SystemExceptions
Category: Language-Exceptions
I am raised when one writes to a read-only object.

6.156.1 SystemExceptions ReadOnlyObject: accessing

description
Answer a textual description of the exception.

6.157 SystemExceptions ShouldNotImplement

Defined in namespace Smalltalk SystemExceptions
Category: Language-Exceptions
I am raised when a method is called that a class wishes that is not called.

6.157.1 SystemExceptions ShouldNotImplement: accessing

description
Answer a textual description of the exception.

6.158 SystemExceptions SubclassResponsibility

Defined in namespace Smalltalk SystemExceptions
Category: Language-Exceptions
I am raised when a method is called whose implementation is the responsibility of concrete subclass.

6.158.1 SystemExceptions SubclassResponsibility: accessing

description
Answer a textual description of the exception.

6.159 SystemExceptions UserInterrupt

Defined in namespace Smalltalk SystemExceptions
Category: Language-Exceptions
I am raised when one presses Ctrl-C.

6.159.1 SystemExceptions UserInterrupt: accessing

description
Answer a textual description of the exception.

6.160 SystemExceptions VMError

Defined in namespace Smalltalk SystemExceptions
Category: Language-Exceptions

I am an error related to the innards of the system.

6.160.1 SystemExceptions VMError: accessing

description

Answer a textual description of the exception.

6.161 SystemExceptions WrongArgumentCount

Defined in namespace Smalltalk SystemExceptions
Category: Language-Exceptions

6.161.1 SystemExceptions WrongArgumentCount: accessing

description

Answer a textual description of the exception.

6.162 SystemExceptions WrongClass

Defined in namespace Smalltalk SystemExceptions
Category: Language-Exceptions

6.162.1 SystemExceptions WrongClass class: signaling

signalOn: anObject mustBe: aClassOrArray

Raise an exception. The given object should have been an instance of one of the classes indicated by aClassOrArray (which should be a single class or an array of classes). Whether instances of subclasses are allowed should be clear from the context, though in general (i.e. with the exception of a few system messages) they should be.

6.162.2 SystemExceptions WrongClass: accessing

description

Answer a textual description of the exception.

messageText

Answer an exception's message text.

validClasses

Answer the list of classes whose instances would have been valid.

validClasses: aCollection
Set the list of classes whose instances would have been valid.

validClassesString
Answer the list of classes whose instances would have been valid, formatted as a string.

6.163 SystemExceptions WrongMessageSent

Defined in namespace Smalltalk SystemExceptions
Category: Language-Exceptions
I am raised when a method is called that a class wishes that is not called. This exception also includes a suggestion on which message should be sent instead

6.163.1 SystemExceptions WrongMessageSent class: signaling

signalOn: selector useInstead: aSymbol
Raise an exception, signaling which selector was sent and suggesting a valid alternative.

6.163.2 SystemExceptions WrongMessageSent: accessing

messageText
Answer an exception's message text.

selector Answer which selector was sent.

selector: aSymbol
Set which selector was sent.

suggestedSelector
Answer a valid alternative to the selector that was used.

suggestedSelector: aSymbol
Set a valid alternative to the selector that was used.

6.164 TextCollector

Defined in namespace Smalltalk
Category: Streams
I am a thread-safe class that maps between standard Stream protocol and a single message to another object (its selector is pluggable and should roughly correspond to #nextPutAll:). I am, in fact, the class that implements the global Transcript object.

6.164.1 TextCollector class: accessing

message: receiverToSelectorAssociation
Answer a new instance of the receiver, that uses the message identified by anAssociation to perform write operations. anAssociation's key is the receiver, while its value is the selector.

new This method should not be called for instances of this class.

6.164.2 TextCollector: accessing

cr Emit a new-line (carriage return) to the Transcript

endEntry Emit two new-lines. This method is present for compatibility with VisualWorks.

next: anInteger put: anObject
Write anInteger copies of anObject to the Transcript

nextPut: aCharacter
Emit aCharacter to the Transcript

nextPutAll: aString
Write aString to the Transcript

show: aString
Write aString to the Transcript

showCr: aString
Write aString to the Transcript, followed by a new-line character

showOnNewLine: aString
Write aString to the Transcript, preceded by a new-line character

6.164.3 TextCollector: printing

print: anObject
Print anObject's representation to the Transcript

printOn: aStream
Print a representation of the receiver onto aStream

6.164.4 TextCollector: set up

message Answer an association representing the message to be sent to perform write operations. The key is the receiver, the value is the selector

message: receiverToSelectorAssociation
Set the message to be sent to perform write operations to the one represented by anAssociation. anAssociation's key is the receiver, while its value is the selector

6.164.5 TextCollector: storing

store: anObject
Print Smalltalk code which evaluates to anObject on the Transcript

storeOn: aStream
Print Smalltalk code which evaluates to the receiver onto aStream

6.165 Time

Defined in namespace Smalltalk
Category: Language-Data types
My instances represent times of the day. I provide methods for instance creation, methods that access components (hours, minutes, and seconds) of a time value, and a block execution timing facility.

6.165.1 Time class: basic (UTC)

utcNow Answer a time representing the current time of day in Coordinated Universal Time (UTC)

utcSecondClock
Answer the number of seconds since the midnight of 1/1/1901 (unlike #secondClock, the reference time is here expressed as UTC, that is as Coordinated Universal Time).

6.165.2 Time class: builtins

primMillisecondClock
Returns the number of milliseconds since midnight.

primSecondClock
Returns the number of seconds to/from 1/1/2000.

timezone Answer a String associated with the current timezone (either standard or daylight-saving) on this operating system. For example, the answer could be 'EST' to indicate Eastern Standard Time; the answer can be empty and can't be assumed to be a three-character code such as 'EST'.

timezoneBias
Specifies the current bias, in minutes, for local time translation for the current time. The bias is the difference, in seconds, between Coordinated Universal Time (UTC) and local time; a positive bias indicates that the local timezone is to the east of Greenwich (e.g. Europe, Asia), while a negative bias indicates that it is to the west (e.g. America)

6.165.3 Time class: clocks

millisecondClock
Answer the number of milliseconds since startup.

millisecondClockValue
Answer the number of milliseconds since startup

millisecondsPerDay
Answer the number of milliseconds in a day

millisecondsToRun: timedBlock
Answer the number of milliseconds which timedBlock took to run

secondClock
Answer the number of seconds since the midnight of 1/1/1901

6.165.4 Time class: initialization

initialize Initialize the Time class after the image has been bootstrapped

update: aspect
Private - Initialize the receiver's instance variables

6.165.5 Time class: instance creation

fromSeconds: secondCount
Answer a Time representing secondCount seconds past midnight

hours: h Answer a Time that is the given number of hours past midnight

hours: h minutes: m seconds: s
Answer a Time that is the given number of hours, minutes and seconds past midnight

minutes: m
Answer a Time that is the given number of minutes past midnight

new Answer a Time representing midnight

now Answer a time representing the current time of day

readFrom: aStream
Parse an instance of the receiver (hours/minutes/seconds) from aStream

seconds: s Answer a Time that is the given number of seconds past midnight

6.165.6 Time: accessing (ANSI for DateAndTimes)

hour Answer the number of hours in the receiver

hour12 Answer the hour in a 12-hour clock

hour24 Answer the hour in a 24-hour clock

minute Answer the number of minutes in the receiver

second Answer the number of seconds in the receiver

6.165.7 Time: accessing (non ANSI & for Durations)

asSeconds Answer 'seconds'.

hours Answer the number of hours in the receiver

minutes Answer the number of minutes in the receiver

seconds Answer the number of seconds in the receiver

6.165.8 Time: arithmetic

addTime: timeAmount
Answer a new Time that is timeAmount seconds after the receiver

printOn: aStream
Print a representation of the receiver on aStream

subtractTime: timeAmount
Answer a new Time that is timeAmount seconds before the receiver

6.165.9 Time: comparing

< aTime Answer whether the receiver is less than aTime

= aTime Answer whether the receiver is equal to aTime

hash Answer an hash value for the receiver

6.166 TokenStream

Defined in namespace Smalltalk
Category: Streams-Collections
I am not a typical part of the Smalltalk kernel class hierarchy. I operate on a stream of characters and return distinct whitespace-delimited groups of characters; I am used to parse the parameters of class-creation methods.

Basically, I parse off whitespace separated tokens as substrings and return them (next). If the entire contents of the string are requested, I return them as an Array containing the individual strings.

6.166.1 TokenStream class: instance creation

on: aString
Answer a TokenStream working on aString

onStream: aStream
Answer a TokenStream working on the collection on which aStream is in turn streaming.

6.166.2 TokenStream: basic

atEnd Answer whether the input stream has no more tokens.

next Answer a new whitespace-separated token from the input stream

6.166.3 TokenStream: write methods

nextPut: anObject
This method should not be called for instances of this class.

6.167 TrappableEvent

Defined in namespace Smalltalk
Category: Language-Exceptions
I am an abstract class for arguments passed to #on:do:... methods in BlockClosure. I define a bunch of methods that apply to CoreExceptions and ExceptionSets: they allow you to create ExceptionSets and examine all the exceptions to be trapped.

6.167.1 TrappableEvent: enumerating

allExceptionsDo: aBlock
Execute aBlock, passing it an Exception for every exception in the receiver.

handles: exception
Answer whether the receiver handles 'exception'.

6.167.2 TrappableEvent: instance creation

, aTrappableEvent
Answer an ExceptionSet containing all the exceptions in the receiver and all the exceptions in aTrappableEvent

6.168 True

Defined in namespace Smalltalk
Category: Language-Data types
I represent truth and justice in the world. My motto is "semper veritatis".

6.168.1 True: basic

& aBoolean
We are true – anded with anything, we always answer the other operand

and: aBlock
We are true – anded with anything, we always answer the other operand, so evaluate aBlock

eqv: aBoolean
Answer whether the receiver and aBoolean represent the same boolean value

ifFalse: falseBlock
We are true – answer nil

ifFalse: falseBlock ifTrue: trueBlock
We are true – evaluate trueBlock

ifTrue: trueBlock
We are true – evaluate trueBlock

ifTrue: trueBlock ifFalse: falseBlock
We are true – evaluate trueBlock

not We are true – answer false

or: aBlock We are true – ored with anything, we always answer true

xor: aBoolean
Answer whether the receiver and aBoolean represent different boolean values

| aBoolean
We are true – ored with anything, we always answer true

6.168.2 True: C hacks

asCBooleanValue
Answer '1'.

6.168.3 True: printing

printOn: aStream
Print a representation of the receiver on aStream

6.169 UndefinedObject

Defined in namespace Smalltalk
Category: Language-Implementation
I have the questionable distinction of being a class with only one instance, which is the object "nil".

6.169.1 UndefinedObject: class creation

metaclassFor: classNameString
Create a Metaclass object for the given class name. The metaclass is a subclass of Class

removeSubclass: aClass
Ignored – necessary to support disjoint class hierarchies

subclass: classNameString
instanceVariableNames: stringInstVarNames classVariableNames: stringOfClassVarNames poolDictionaries: stringOfPoolNames category: categoryNameString Define a fixed subclass of the receiver with the given name, instance variables, class variables, pool dictionaries and category. If the class is already defined, if necessary, recompile everything needed.

variableByteSubclass: classNameString
instanceVariableNames: stringInstVarNames classVariableNames: stringOfClassVarNames poolDictionaries: stringOfPoolNames category: categoryNameString Define a byte variable subclass of the receiver with the given name, instance variables, class variables, pool dictionaries and category. If the class is already defined, if necessary, recompile everything needed.

variableSubclass: classNameString
instanceVariableNames: stringInstVarNames classVariableNames: stringOfClassVarNames poolDictionaries: stringOfPoolNames category: categoryNameString Define a variable pointer subclass of the receiver with the given name, instance variables, class variables, pool dictionaries and category. If the class is already defined, if necessary, recompile everything needed.

variableWordSubclass: classNameString
instanceVariableNames: stringInstVarNames classVariableNames: stringOfClassVarNames poolDictionaries: stringOfPoolNames category: categoryNameString Define a word variable subclass of the receiver with the given name, instance variables, class variables, pool dictionaries and category. If the class is already defined, if necessary, recompile everything needed.

6.169.2 UndefinedObject: class creation - alternative

subclass: classNameString classInstanceVariableNames: stringClassInstVarNames instanceVariableNames: stringInstVarNames classVariableNames: stringOfClassVarNames poolDictionaries: stringOfPoolNames
Don't use this, it is only present to file in from IBM Smalltalk

subclass: classNameString instanceVariableNames: stringInstVarNames classVariableNames: stringOfClassVarNames poolDictionaries: stringOfPoolNames
Don't use this, it is only present to file in from IBM Smalltalk

variableByteSubclass: classNameString classInstanceVariableNames: stringClassInstVarNames instanceVariableNames: stringInstVarNames classVariableNames: stringOfClassVarNames poolDictionaries: stringOfPoolNames
Don't use this, it is only present to file in from IBM Smalltalk

variableByteSubclass: classNameString instanceVariableNames: stringInstVarNames classVariableNames: stringOfClassVarNames poolDictionaries: stringOfPoolNames
Don't use this, it is only present to file in from IBM Smalltalk

variableSubclass: classNameString classInstanceVariableNames: stringClassInstVarNames instanceVariableNames: stringInstVarNames classVariableNames: stringOfClassVarNames poolDictionaries: stringOfPoolNames
Don't use this, it is only present to file in from IBM Smalltalk

variableSubclass: classNameString instanceVariableNames: stringInstVarNames classVariableNames: stringOfClassVarNames poolDictionaries: stringOfPoolNames
Don't use this, it is only present to file in from IBM Smalltalk

variableWordSubclass: classNameString classInstanceVariableNames: stringClassInstVarNames instanceVariableNames: stringInstVarNames classVariableNames: stringOfClassVarNames poolDictionaries: stringOfPoolNames
Don't use this, it is only present to file in from IBM Smalltalk

variableWordSubclass: classNameString instanceVariableNames: stringInstVarNames classVariableNames: stringOfClassVarNames poolDictionaries: stringOfPoolNames
Don't use this, it is only present to file in from IBM Smalltalk

6.169.3 UndefinedObject: CObject interoperability

free Do nothing, a NULL pointer can be safely freed.

6.169.4 UndefinedObject: dependents access

addDependent: ignored
Refer to the comment in Object|dependents.

release Nil release is a no-op

6.169.5 UndefinedObject: printing

printOn: aStream
Print a representation of the receiver on aStream.

6.169.6 UndefinedObject: storing

storeOn: aStream
Store Smalltalk code compiling to the receiver on aStream.

6.169.7 UndefinedObject: testing

ifNil: nilBlock
Evaluate nilBlock if the receiver is nil, else answer nil

ifNil: nilBlock ifNotNil: notNilBlock
Evaluate nilBlock if the receiver is nil, else evaluate notNilBlock, passing the receiver.

ifNotNil: notNilBlock
Evaluate notNilBlock if the receiver is not nil, passing the receiver. Else answer nil

ifNotNil: notNilBlock ifNil: nilBlock
Evaluate nilBlock if the receiver is nil, else evaluate notNilBlock, passing the receiver.

isNil Answer whether the receiver is the undefined object nil. Always answer true.

notNil Answer whether the receiver is not the undefined object nil. Always answer false.

6.170 ValueAdaptor

Defined in namespace Smalltalk
Category: Language-Data types
My subclasses are used to access data from different objects with a consistent protocol. However, I'm an abstract class.

6.170.1 ValueAdaptor class: creating instances

new We don't know enough of subclasses to have a shared implementation of new

6.170.2 ValueAdaptor: accessing

value Retrive the value of the receiver. Must be implemented by ValueAdaptor's subclasses

value: anObject
Set the value of the receiver. Must be implemented by ValueAdaptor's subclasses

6.170.3 ValueAdaptor: basic

printOn: aStream
Print a representation of the receiver

6.171 ValueHolder

Defined in namespace Smalltalk
Category: Language-Data types

I store my value in a variable, and know whether I have been initialized or not. If you ask for my value and I have not been initialized, I suspend the process until a value has been assigned.

6.171.1 ValueHolder class: creating instances

new Create a ValueHolder whose starting value is nil

null Answer the sole instance of NullValueHolder

with: anObject
Create a ValueHolder whose starting value is anObject

6.171.2 ValueHolder: accessing

value Get the value of the receiver.

value: anObject
Set the value of the receiver.

6.171.3 ValueHolder: initializing

initialize Private - set the initial value of the receiver

6.172 VersionableObjectProxy

Defined in namespace Smalltalk
Category: Streams-Files

I am a proxy that stores additional information to allow different versions of an object's representations to be handled by the program. VersionableObjectProxies are backwards compatible, that is you can support versioning even if you did not use a VersionableObjectProxy for that class when the object was originarily dumped. VersionableObjectProxy does not support classes that changed shape across different versions. See the method comments for more information.

6.172.1 VersionableObjectProxy class: saving and restoring

loadFrom: anObjectDumper
Retrieve the object. If the version number doesn't match the -#binaryRepresentationVersion answered by the class, call the class' #convertFromVersion:withFixedVariables:instanceVariables:for: method. The stored version number will be the first parameter to that method (or nil if

the stored object did not employ a VersionableObjectProxy), the remaining parameters will be respectively the fixed instance variables, the indexed instance variables (or nil if the class is fixed), and the ObjectDumper itself. If no VersionableObjectProxy, the class is sent #nonVersionedInstSize to retrieve the number of fixed instance variables stored for the non-versioned object.

6.172.2 VersionableObjectProxy: saving and restoring

dumpTo: anObjectDumper
Save the object with extra versioning information.

6.173 Warning

Defined in namespace Smalltalk
Category: Language-Exceptions
Warning represents an 'important' but resumable error.

6.173.1 Warning: exception description

description
Answer a textual description of the exception.

6.174 WeakArray

Defined in namespace Smalltalk
Category: Collections-Weak
I am similar to a plain array, but my items are stored in a weak object, so I track which of them are garbage collected.

6.174.1 WeakArray class: instance creation

new: size Private - Initialize the values array; plus, make it weak and create the ByteArray used to track garbage collected values

6.174.2 WeakArray: accessing

aliveObjectsDo: aBlock
Evaluate aBlock for all the elements in the array, excluding the garbage collected ones. Note: a finalized object stays alive until the next collection (the collector has no means to see whether it was resuscitated by the finalizer), so an object being alive does not mean that it is usable.

at: index Answer the index-th item of the receiver, or nil if it has been garbage collected.

at: index put: object
Store the value associated to the given index; plus, store in nilValues whether the object is nil. nil objects whose associated item of nilValues is 1 were touched by the garbage collector.

atAll: indices put: object
Put object at every index contained in the indices collection

atAllPut: object
Put object at every index in the receiver

clearGCFlag: index
Clear the 'object has been garbage collected' flag for the item at the given index

do: aBlock
Evaluate aBlock for all the elements in the array, including the garbage collected ones (pass nil for those).

isAlive: index
Answer whether the item at the given index is still alive or has been garbage collected. Note: a finalized object stays alive until the next collection (the collector has no means to see whether it was resuscitated by the finalizer), so an object being alive does not mean that it is usable.

size Answer the number of items in the receiver

6.174.3 WeakArray: conversion

asArray Answer a non-weak version of the receiver

deepCopy Returns a deep copy of the receiver (the instance variables are copies of the receiver's instance variables)

shallowCopy
Returns a shallow copy of the receiver (the instance variables are not copied)

species Answer Array; this method is used in the #copyEmpty: message, which in turn is used by all collection-returning methods (collect:, select:, reject:, etc.).

6.174.4 WeakArray: loading

postLoad Called after loading an object; must restore it to the state before 'preStore' was called. Make it weak again

6.175 WeakIdentitySet

Defined in namespace Smalltalk
Category: Collections-Weak
I am similar to a plain identity set, but my keys are stored in a weak array; I track which of them are garbage collected and, as soon as I encounter one of them, I swiftly remove all the garbage collected keys

6.176 WeakKeyIdentityDictionary

Defined in namespace Smalltalk
Category: Collections-Weak

I am similar to a plain identity dictionary, but my keys are stored in a weak array; I track which of them are garbage collected and, as soon as I encounter one of them, I swiftly remove all the associations for the garbage collected keys

6.177 WeakKeyLookupTable

Defined in namespace Smalltalk
Category: Collections-Weak

I am similar to a plain LookupTable, but my keys are stored in a weak array; I track which of them are garbage collected and, as soon as I encounter one of them, I swiftly remove all the associations for the garbage collected keys

6.177.1 WeakKeyLookupTable class: instance creation

new: anInteger

Answer a new instance of the receiver with the given size

6.177.2 WeakKeyLookupTable: rehashing

rehash Rehash the receiver

6.178 WeakSet

Defined in namespace Smalltalk
Category: Collections-Weak

I am similar to a plain set, but my items are stored in a weak array; I track which of them are garbage collected and, as soon as I encounter one of them, I swiftly remove all.

6.178.1 WeakSet class: instance creation

new: anInteger

Answer a new instance of the receiver with the given size

6.178.2 WeakSet: rehashing

rehash Rehash the receiver

6.179 WeakValueIdentityDictionary

Defined in namespace Smalltalk
Category: Collections-Weak

I am similar to a plain identity dictionary, but my values are stored in a weak array; I track which of the values are garbage collected and, as soon as one of them is accessed, I swiftly remove the associations for the garbage collected values

6.180 WeakValueLookupTable

Defined in namespace Smalltalk
Category: Collections-Weak

I am similar to a plain LookupTable, but my values are stored in a weak array; I track which of the values are garbage collected and, as soon as one of them is accessed, I swiftly remove the associations for the garbage collected values

6.180.1 WeakValueLookupTable: hacks

at: key ifAbsent: aBlock

Answer the value associated to the given key, or the result of evaluating aBlock if the key is not found

at: key ifPresent: aBlock

If aKey is absent, answer nil. Else, evaluate aBlock passing the associated value and answer the result of the invocation

includesKey: key

Answer whether the receiver contains the given key.

6.180.2 WeakValueLookupTable: rehashing

rehash Rehash the receiver

6.181 WordArray

Defined in namespace Smalltalk
Category: Collections-Sequenceable

I am similar to a plain array, but my items are 32-bit integers.

6.182 WriteStream

Defined in namespace Smalltalk
Category: Streams-Collection

I am the class of writeable streams. I only allow write operations to my instances; reading is strictly forbidden.

6.182.1 WriteStream class: instance creation

on: aCollection
Answer a new instance of the receiver which streams on aCollection. Every item of aCollection is discarded.

with: aCollection
Answer a new instance of the receiver which streams from the end of aCollection.

with: aCollection from: firstIndex to: lastIndex
Answer a new instance of the receiver which streams from the firstIndex-th item of aCollection to the lastIndex-th. The pointer is moved to the last item in that range.

6.182.2 WriteStream: accessing

size Answer how many objects have been written

6.182.3 WriteStream: accessing-writing

nextPut: anObject
Store anObject as the next item in the receiver. Grow the collection if necessary

6.182.4 WriteStream: positioning

emptyStream
Extension - Reset the stream

position: anInteger
Skip to the anInteger-th item backwards in the stream. Fail if anInteger>self position. The stream is truncated after the pointer

skip: anInteger
Skip (anInteger negated) items backwards in the stream. Fail if anInteger>0. The stream is truncated after the pointer

6.183 ZeroDivide

Defined in namespace Smalltalk
Category: Language-Exceptions
A ZeroDivide exception is raised by numeric classes when a program tries to divide by zero. Information on the dividend is available to the handler.

6.183.1 ZeroDivide class: instance creation

dividend: aNumber
Create a new ZeroDivide object remembering that the dividend was aNumber.

new Create a new ZeroDivide object; the dividend is conventionally set to zero.

6.183.2 ZeroDivide: accessing

dividend Answer the number that was being divided by zero

6.183.3 ZeroDivide: description

description
Answer a textual description of the exception.

7 Future directions for GNU Smalltalk

Presented below is the set of tasks that I feel need to be performed to make GNU Smalltalk a more fully functional, viable system. They are presented in no particular order; other tasks are listed in the '`TODO`' file, in the main distribution directory.

I would *very much* welcome any volunteers who would like to help with the implementation of one or more of these tasks. Please write at help-smalltalk@gnu.org if you are interested in adding your efforts to the GNU Smalltalk project.

Tasks:

- Port software to GNU Smalltalk. The class library has proven to be quite robust; it should be easy to port packages (especially free ones!) to GNU Smalltalk if the source dialect is reasonably ANSI-compliant. One area which might give problems is exception handling and namespaces.
- Port to other computers/operating systems. The code thus far has shown itself to be relatively portable to various machines and Unix derivatives. The architecture must support 32 or 64 bit pointers the same size as a long integers.
- Comment the C code more thoroughly. The C source code could definitely stand better commenting.
- Modify the Delay class primitive so that it does not fork a new process each time it is called; this involves using a pipe. I want to do it somewhen, but if you do it before me, please tell me.
- Add more test cases to the test suite. It is getting larger, but one good way to help, and learn some Smalltalk in the process, is still to add files and tests to the test suite directory. Ideally, the test suite would be used as the "go/nogo" gauge for whether a particular port or improvement to GNU Smalltalk is really working properly.

Class index

(Index is nonexistent)

Method index

(Index is nonexistent)

Selector cross-reference

(Index is nonexistent)

Table of Contents