Little Smalltalk Reference Manual

Vincent Manis
The University of British Columbia
Department of Computer Science
Vancouver, BC Canada V62 1W5

Draft Version—February 22, 1999
Corrected at Northeastern University
by COM1204 and COM3230 students

Contents

Chapter 1

Introduction

The Little Smalltalk language was developed by Timothy Budd as a vehicle for teaching about object-oriented programming. The first version of the system[1] was useful, but suffered from a number of deficiencies. Budd subsequently released a revised version, Little Smalltalk 3.04, with a number of improvements.

At UBC, further work on Little Smalltalk has centered upon fixing bugs and improving error recovery. A number of small language changes were made, mostly to improve readability and maintain compatibility with other Smalltalk implementations, notably Smalltalk/V.[2]

1.1 Concepts

In Little Smalltalk, everything is an *object*. When we say "everything," we mean *everything*: even numbers and booleans are objects. All objects have three characteristics:

- *messages*, each of which commands it to do some thing. A student object might respond to a message such as `name`, which commands it to return the name of the student; a bank account object might respond to the message `deposit:` n, which tells it to record a deposit of \$$n$.

- *state*: each object "remembers" its internal state. A bank account object might remember its balance, as well as the name and address of the customer; a student object might remember the student's name, year, and marks. Even an integer object remembers which integer it is.

- *behaviour*: each object responds to a set of

- *identity*: different objects are indeed different, and changes to the state of one are not reflected in the state of another. Thus even if two bank accounts happen to have the same state (e.g., Jane Doe has opened two accounts both of which happen to have a balance of \$1310.72), deposits to one do not result in the balance of the other changing.

Rather than specify the behaviour for each object separately (which would be an enormous task, especially when you realize that there are several billion numbers which Smalltalk

[1] Budd, T. *A Little Smalltalk*, Addison-Wesley, 1987.

[2] "Smalltalk/V" is a trademark of Digitalk, Inc.

can manipulate), objects are grouped into *classes*. All objects of a given class respond to the same set of messages, in essentially the same way.[3]

A class definition contains a set of *methods*, each of which contains code for responding to one message. For example, a bank account class would contain a method for responding to the message `deposit:` n. When we send an account object the message `deposit: 200`, the object will refer to its class definition, which does indeed contain a matching method, which is then executed.

Within a class definition, it is sometimes useful to know which object received the message which invoked a given method. For example, suppose that we are computing interest for a customer; the best way of crediting the interest to the customer's account is to send a `deposit:` message. But to which account should the message be sent? To the account which is computing interest. We can therefore say `self deposit: n`.

Sometimes, it is convenient to have *subclasses* of a given class. For example, we might have *savings* accounts, which support all of the other bank account operations, but do not allow withdrawals. Rather than write all of the bank account methods over again for this new class of accounts, we can declare that the savings account class is a subclass of the account class. The only method which would be contained in the savings account class definition is `withdraw:` n; the code for this method would refuse the transaction. All other methods would be *inherited* from the superclass. Thus a `deposit:` 50 message sent to a savings account would find no matching method in the savings account class definition; however, a search of the class definition for the superclass, bank account, would turn up a `deposit:` n method, which would then be executed.

All classes inherit, directly or indirectly, from the class `Object`. This class includes a number of methods which are useful for all objects, such as printing the object out.[4]

Not all methods of a class definition are intended to be used from client code. Sometimes, we will just want a method for use in the implementation of another method. In this case, we will write a *private method*. Smalltalk does not distinguish between private and public methods,[5] but, by convention, private methods are flagged with the comment `Private` at the beginning of a method. We can invoke a private method by sending a message to `self`.

The state of an object is represented by the values of its *instance variables*, a set of variables which exist separately for each object. The values of these variables are not accessible outside the class definition for the object,[6] which means that we must provide methods for accessing and mutating variables.

By convention, Smalltalk programmers divide instance variables into three groups:

- *public and mutable*: methods exist both for accessing and mutate the variables. It may not be possible to set an instance variable to a particular value (e.g., we may be able to deposit and withdraw, but not simply to set the balance to a given value), but if any mutation is permitted, then we consider the instance variable to be mutable.

[3] "Essentially" here refers to the fact that some objects may have states which prevent certain actions from being taken. For example, a customer with \$500 on deposit would be permitted to make a \$100 withdrawal, but a customer with only \$75 would not.

[4] The default printing action is to display the class name, but this method is overridden in a number of system classes, such as strings and numbers. You can, of course, override it in your own classes, so as to define your own print format for an object.

[5] An unfortunate oversight in the design of the language.

[6] In particular, they are not accessible in the class definition of a subclass.

- *public and immutable*: we can access the instance variable, but no method exists for changing it. For example, the value of an integer object is immutable: `5` can't suddently decide to become `6`. Attributes such as Social Insurance Number, which we might use to identify a particular employee, would also be immutable.
- *private*: there is no explicit evidence, outside the class definition, that this instance variable exists. For example, we might store the salary owing to an hourly employee in two instance variables, one recording straight time and the other recording overtime. However, all we can ask the employee object is the total amount of salary owing.

1.2 Using the system

Normally, you will prepare your program as a set of files, which can be loaded into the system. Once the system has been started, and your program has been loaded, the system will enter a "read-evaluate-print" loop, in which a statement (or series of statements) will be read from the keyboard, compiled into bytecodes,[7], and executed. After execution, the resulting object is then sent a `print` message.

The system signals its readiness to execute a command by displaying a prompt, `>`. Here is an example of a simple dialogue:

```
> 2+2
4
> 'The cat in the hat' copyFrom: 5 to: 7
cat
```

When you are finished, exit by typing an end-of-file.[8]

A subsequent chapter describes errors.

[7]Little Smalltalk uses a "virtual machine" approach, in which Smalltalk code is represented as an array of instructions for a hypothetical computer.

[8]`Control-D` for UNIX, `Control-Z Return` for DOS.

Chapter 2

The Little Smalltalk Language

This chapter describes the syntax and semantics of the Little Smalltalk language. Note that, unlike many traditional languages, most of what a Little Smalltalk sees is not the language as such, but the class library, a set of modules which define objects which are useful in most programs. See the next chapter for a description of the class library.

This chapter describes the syntax (rules for writing programs) and semantics (meaning of the programs) for Little Smalltalk.

2.1 Notation

As with many other language manuals, we use a somewhat formal notation to describe the syntax of Little Smalltalk. The notation used here is called "Extended Backus-Naur Form," or EBNF.[1] EBNF consists of a set of *rules*, each of which defines a particular syntax notation.

For example, consider the somewhat derisive language used by a noisy sports fan. Such a fan might be heard to yell `Get a ref!` each time the refereee made a ruling s/he didn't agree with, and `Sha-na-na, hey-hey, good-bye!` each time the opposing team did something s/he didn't like. If the fan gets too offensive, management might ask him/her to leave. We might describe this "language" with the following rules:

```
sports-language:    {taunt} [ejection].
taunt:              get-a-ref | sha-na-na.
get-a-ref:          "Get a ref!".
sha-na-na:          "Sha-na-na, hey-hey, good-bye!"
ejection:           "I am afraid I must ask you to leave."
```

This set of rules says that a "program" in the sports language consists of zero or more taunts,[2] followed possibly by management ejecting the fan (ejection is optional). Each taunt is either a `get-a-ref` or a `sha-na-na`, each of which is, in turn, a specific string.[3]

[1]The original version of this notation was developed by John Backus and Peter Naur for describing the language ALGOL 60, in 1960. Niklaus Wirth extended Backus-Naur Form by including a number of features found in IBM manuals for describing programming languages. Wirth, with characteristic modesty, refrained from calling his system "WBNF."

[2]"taunt n (1529):a sarcastic challenge or insult"

[3]Needless to say, this is only a fanciful example. We do not condone hooliganism or offensive behaviour at sporting events.

Formally, an EBNF specification for a language consists of a set of rules, each of which has a left-hand-side and a right-hand-side. The left-hand-side is a single symbol, and the right-hand side is a set of symbols as follows:

- a symbol refers to the left-hand-side of another rule. Anything which is valid according to that rule may be written here.
- braces `{}` mean zero or more repetitions of what appears inside the braces.
- square brackets means that what is inside the brackets is optional (may or may not appear).
- the pipe symbol `|` means "or."
- parentheses are used to make grouping of braces and pipes explicit.
- things which are to be written as is are enclosed in quotation marks.

Since EBNF is not intended to be interpreted by a computer, but only by a human, we don't have to be quite as precise about the syntax as we would be if we were writing an evaluator for EBNF.[4]

Here's a description of EBNF, written in EBNF:

```
EBNF-specification: {rule}.
rule:               left-side ":" right-side "."
left-side:          symbol.
right-side:         alternative { "|" alternative}.
alternative:        {item}.
item:               symbol | quoted-string | repetition |
                    group | option.
repetition:         "{" {item} "}".
group:              "(" {item} ")".
option:             "[" {item} "]".
symbol:             anything which looks like a symbol,
                    perhaps with embedded hyphens.
quoted-string:      "anything (quote marks are doubled)"
```

Note that the `symbol` and `quoted-string` rules aren't in EBNF themselves. It is very common to find this phenomenon in EBNF specifications: the reader is assumed to be smart enough to figure out what is valid for such a rule.

Unfortunately, no comparably simple notation exists for describing the semantics of programming languages. We shall rely on English for explaining the semantics of Little Smalltalk.

2.2 Token syntax

A *token* is a single Little Smalltalk language element, e.g., a number, string, special character, etc.

[4] In fact, many compilers are written partly in languages based upon EBNF.

Comments may be written anywhere, surrounded in quote marks. **Bug:** Comments outside the `Methods` part of a source file must be complete on one line.

Some characters require special attention:

- the period is used as a decimal point in a number as well as a statement terminator. Write floating-point numbers with digits on both sides of the period in order to avoid ambiguity, e.g., `3.5`.
- the pipe `|` is used for three [sic!] different purposes:
  1. to mark the end of a method. In this case, it always appears in column 1 of a line.
  2. to flag the temporary variables of a method. In this case, it does not appear in column 1, but is surrounded by spaces.
  3. as a binary message selector.

When you type a statement into Little Smalltalk, pressing RETURN will result in the statement being sent to Little Smalltalk for evaluation. To split such a statement into multiple lines, end each line except for the last with a backslash, \\.

Other than as specified in the previous paragraphs, spaces, and newlines may be inserted at will between adjacent tokens, without changing the meaning of a program.

2.3 Expressions

Expressions are used to compute a value. For example, the expression `2+2` evaluates to `5`, and the expression `'confound' copyFrom: 2 to: 3` evaluates to `'on'`.

2.3.1 Literals

A number of types of objects may be written using reasonably standard notation.[5] Numbers, characters, strings, symbols, and arrays may be written literally. Two subclasses of arrays, byte arrays and strings, have their own constant representation.

The term "literal" is used to refer to such a constant value in a program.

```
literal:                number | symbol | string |
                        character | bytearray | array .
```

There are 6 types of literals built in to Little Smalltalk: numbers, symbols, strings, character constants, byte array constants, and array constants.

Numbers

```
number:                 integer | floating-number.
integer:                [base] digit {digit}.
base:                   digit {digit} "r".
floating-number:        digit {digit} "." digit {digit} [exponent].
exponent:               "e" ("+" | "-") digit {digit}
```

[5] In this sense, the classes of these objects are handled specially by the compiler: classes such as `Integer` must be present in the system for it to function at all. However, you can normally ignore this fact.

Numbers may be either integers or floating-point numbers. An integer is written as a string of digits, possibly preceded by a base specification. The base specification is a number between `2` and `36` which specifies the base in which the number is represented: bases larger than `10` use the "digits" `A` through `Z`. **Bug:** Bases other than 10 don't seem to work properly.

There is no limitation (other than the total memory of the computer) on the size of a number; numbers larger than a threshold value (presently `16384`) are treated as instances of class `LongInteger`, but this generally does not make any difference to the programmer. **Bug:** Long integers sometimes print with extra leading zeroes. Literals typed in as long integers are sometimes treated as floating-point numbers.

Floating-point numbers have decimal places, and maybe an exponent, representing a scale-factor as a power of 10. Because of the nature of binary arithmetic, floating-point numbers are not represented exactly. Note that a floating-point number must be written with digits on both sides of the decimal point; otherwise, the compiler will take the decimal point as being a statement terminator.

There is no notation for fractions, but an integer sent the message `/ anInteger` will return a fraction.

Example

```
2 -35
2r101 16rBAD   "These don't work properly"
2.718281828459045 4.5e3  -9.7e-2
```

Symbols

A symbol is an object which only has identity. Different symbols represent different objects, but have little other behaviour. Upper and lower case are considered different.

```
symbol:                 "#" symbol-char {symbol-char}.
symbol-char:            "Any character other than period,
                        parentheses, or square brackets."
```

Example

```
#smalltalk #Smalltalk  "These two symbols are different"
#+ #at:put:
#this-is-a-symbol-too
```

Strings

A string is a sequence of characters enclosed in *single* apostrophes. A single apostrophe inside a string is represented as two consecutive apostrophes.

```
string:                 "'" {string-char} "'".
string-char:            non-apostrophe-char | "''".
non-apostrophe-char:    "any printable character except an
                        apostrophe".
```

Character

A single character is represented as a dollar sign followed by a single printable character.

```
character:                "$" any-char.
any-char:                 "any character at all".
```

For non-printable characters, you will probably prefer to construct the character by sending an `asChar` message to the integer whose character code you desire, e.g., `7 asChar` to produce the character whose code is `7`.

Example

```
$a $$ $      "The third example is the character ‘space’"
```

Byte arrays

You'll never need byte arrays.

Arrays

It is sometimes extremely convenient to be able to write an entire array explicitly.

```
array:                    "#" "(" {literal} ")".
```

Inside an array constant, \# characters may be omitted on symbols, arrays, and byte arrays.

Example

```
#(1 2 3 4)
#(This is a demo)   "An array of 4 symbols."
#((1 2 3) (4 5 6))  "An array of 2 elements, each of
                     which is an array of 3 elements."
```

2.3.2 Variables and pseudo-variables

Little Smalltalk has two different kinds of variables:

- Global variables are accessible throughout the whole program. Needless to say, an overreliance upon global variables will impair the readability of a program. Global variable names are spelled beginning with a capital letter.
- Local variables are accessible only in a given class, or even in a given method or block. Local variable names are spelled beginning with a lower-case letter.

```
identifier:               local-identifier | global-identifier.
local-identifier:         lc-letter {letter | digit}.
global-identifier:        uc-letter {letter | digit}.
letter:                   lc-letter | uc-letter.
lc-letter:                "a" | "b" | ... | "z".
uc-letter:                "A" | "B" | ... | "Z".
```

Bug: Smalltalk does not presently enforce the spelling restriction described in the previous paragraph.

Scoping is lexical in Little Smalltalk.

There are also a number of "pseudo-variables:"

- `true` and `false` represent the Boolean values.
- `nil` is the name for the only instance of class `UndefinedObject`: newly created variables are initialized to `nil`, and, by convention `nil` is used to mark the end of a list or other linked structure.
- `self` represents the current object, i.e., the receiver of the current message. It does not make sense to use `self` in code typed in to the interpreter.
- `super` also represents the current object, but when a message is sent to `super`, the hunt for the method starts in the superclass of the object.
- `selfProcess` is used for the multi-processing features of Little Smalltalk, and won't be described further here.
- `smalltalk` is the sole instance of class `Smalltalk`, and is used for various system-related tasks.

2.3.3 Message expressions

More complicated expressions involve the sending of messages. There are three kinds of messages:

- a *unary message* consists of a single symbol, with no arguments.
- a *binary message* consists of a "binary selector," along with a single argument. A binary selector is one or two special characters, which is any punctuation character, excluding characters such as parentheses, braces, brackets, the period, the sharp, and other characters which are treated specially by Little Smalltalk. Note that while we would think of the selector `+` as meaning addition, the exact meaning of a `+` message to an object would be up to the person writing the class definition for that object.
- a *keyword message* consists of a series of keywords, each ending with a colon, and corresponding arguments.

Normally, unary messages are sent first, then binary messages, and finally keyword messages. Parentheses may be used to force other groupings.

```
expression:           primary {keyword: binary}
keyword:              local-identifier ":"
binary:               primary binary-selector unary
binary-selector:      operator-char [operator-char]
unary:                primary {unary-selector}
unary-selector:       local-identifier
primary:              variable | literal | block |
                      primitive | "(" cascade ")"
```

Blocks, cascades and primitives will be explained later. For now, it is sufficient to know that any expression is a cascade, which is what lets us use parentheses.

Example

```
3 negated                       "A unary message"
2 + 2                           "A binary message"
5 between: 3 and: 7             "A keyword message."
2 + 2 negated                   "Evaluates to 0."
(2 + 2) negated                 "Evaluates to -4."
-2 between: 2+2 negated and: 2 "Evaluates to false."
```

In the third example, the message being sent was `between:and:`, with the arguments `3` and `7`. In the fourth example, the `negated` message, as a unary, is evaluated before the binary `+` message. In the last example, `-2` is an integer constant. First the `negated` message is sent, and then the `+` message; finally the keyword message `between:and:` is sent, with arguments `0` and `2`.

Keep in mind that the compiler decodes keyword messages by searching for the longest string of keywords. Thus, for example, if an object `Foo`'s class provides `name:` and `addr:` methods, attempting to evaluate

```
Foo name: 'Septimius Severus'
    addr: 'LXVII Via Palatina, Rome'
```

(on one line) will result in an error message (at run-time), because the compiler will think you're trying to send the `name:addr:` message, which is not implemented. (If you want to send a string of messages to the same object, use a cascade.)

2.3.4 Cascades

Often, it is convenient to send a sequence of messages to the same receiver. This is particularly useful during initialization.

```
cascade:                expression {";" message}.
message:                unary-selector |
                        binary-selector unary |
                        keyword binary {keyword binary}.
```

A cascade, therefore, starts with an expression. You can send as many messages as you like to the receiver of that expression.

Example

```
Foo
   name: 'Ebenezer Q. Scranchley';
   empNum: 12345;
   rank: #Programmer-3
```

In this example, we are assuming that `Foo` is an instance of class `Employee`; this class is assumed to respond to methods `name:`, `empNum:`, and `rank:`, among many others. Each of these messages is sent to `Foo` in turn.

The usefulness of a cascade becomes apparent when we realize that the receiver can itself be specified by an expression (though that expression will have to be in parentheses).

In comparing the example to the above syntax rules, it is helpful to note that the `expression` is `Foo name: 'Ebenezer Q. Scranchley'`, and that two `messages` are sent, `empNum: 12345` and `rank: \#Programmer-3`.

2.3.5 Blocks

A block is what would be called in other languages a "procedure." It has arguments, and computes a value.

```
block:                  "[" {arguments} sequence "]".
arguments:              colon-id {colon-id} "|".
colon-id:               ":" local-identifier.
```

The arguments are the names for the formal arguments of the block. If no arguments are present, then the pipe afterwards is omitted.

A `sequence` is a set of statements (discussed below).

Note that the actual argument name in the sequence does not include the colon.[6]

Example

```
[5]                 "A procedure which always returns 5."
['Foo' print]       "A procedure which prints 'Foo'."
[:x | x+1]          "A procedure which returns its
                     argument plus 1."
[:a | a print. a negated]
                    "Prints its argument, and returns
                     the negative of the argument."
```

2.3.6 Primitives

A *primitive* is an expression which invokes a method within the Little Smalltalk interpreter. Approximately 150 primitive methods are present within the interpreter, and an experienced C programmer can easily add custom primitive methods.

You will *never* need to invoke a primitive from your code (because Smalltalk's class library does so when necessary), so, therefore, this section is probably of no practical interest to you, unless you add your own primitive methods to the interpreter.

```
primitive:              "<" integer {primary} ">".
```

The integer will be a number between 0 and 255 to select the primitive. The consecutive `primaries` are the arguments of the primitive method.

2.4 Statements and sequences

The bodies of blocks and methods are statement sequences.

```
sequence:               statement {"." statement}
statement:              assignment | expression | return
```

[6]Colons must precede the argument names so as to avoid ambiguity between blocks with and without arguments.

Note that periods are statement *separators*, not statement *terminators*. Placing a period after the last statement will result in syntax error messages.

The value returned by the statement sequence is the value of the last statement. **Example**

```
a <- 3. 'Done' print. 37
```

The above sequence sets `a` to 3, prints `Done`, and then returns the value `37`.

2.4.1 Assignments

An *assignment* is a statement which mutates a variable.

```
assignment:                {identifier <-} expression.
```

The identifier can be local or global (though mutating global variables without care is poor programming style).

For compatibility with earlier versions of Little Smalltalk, `<-` is accepted as equivalent to `:=`. New code should use `:=`. However, in the version we're using only `<-` is accepted.

Example

```
a <- 3 sqrt
b <- c <-
   Employee new
      name: 'Rodolpho M. Dunkelhelm'
      empNo: 12345
```

In the second example, `b` and `c` are assigned the same value. This means that they will *share* storage: changes to the state of one will result in changes to the state of the other.

2.4.2 Expression statements

An expression may be used as a statement. If the expression is the last statement in a sequence, the value of the expression is the value of the sequence. Otherwise, the value of the expression is thrown away (and thus, unless the expression has side-effects, such as mutating memory or displaying something on the screen, it is redundant).

2.4.3 Return statements

A return statement gives an expression whose value is to be returned as the result of the *method* it is in.

```
return:                    "^" expression
```

A return statement cannot be other than the last expression in a sequence (and thus is never followed by a period). Note that a return statement does not give a value for the block, but for the whole method it is in.

Example

```
(x < 5) ifTrue: [^7] ifFalse: ['x too large' print].
a <- a+1.
^9
```

This nonsensical example returns `7` from the code if `x<5`. Otherwise, a message is printed, `a` is incremented, and `9` is returned.

If control "drops off the bottom" of a method (i.e., no return statement is executed) it is as if `^self` had been executed at the end.

2.5 Classes

Normally, a Little Smalltalk programmer will develop a series of class descriptions which are stored in a *source file*, and then loaded into Little Smalltalk for execution. A source file consists of a number of *headers*, each of which defines a class, followed by a number of method packages.

2.5.1 Class header

```
class-header:           "Class" class-name
                        superclass {instance-var}.
class-name:             global-identifier.
superclass:             global-identifier.
instance-var:           local-identifier.
```

Example

```
Class Point Object xCoord yCoord
```

2.5.2 Method package

```
method-package:         method-header "[" {method} "]".
method-header:          "Methods" class-name category.
category:               string.
```

The category is essentially a comment; you can have many method packages for the same class, and the category is intended to give you a way of grouping methods in a coherent way.

Example

```
Methods Point 'creation'
[
   new
      xCoord <- 0.
      yCoord <- 0
]
Methods Point 'accessing & mutating'
[
   x
      ^xCoord
|
```

```
   y
      ^yCoord
|
   x: newX
      xCoord <- newX
|
   y: newY
      yCoord <- newY
]
```

There is never a need to use multiple methods packages for the same class, unless the number of methods is so large that you would like to break it across multiple source files, or unless you wish to add new methods to system-defined classes.

The `category` is used to tell the reader something about the rationale for grouping these methods into this package. It is often something such as `'creation'`, `'printing'`, or `'mutating'`, which tells you something about the methods in that package. In the common case where all of the methods of a class are in one package, use `'all'`.

2.5.3 Method

```
method:             method-pattern [temporaries] sequence.
method-pattern:     unary-pattern | binary-pattern |
                    keyword-pattern.
unary-pattern:      identifier.
binary-pattern:     binary-selector identifier.
keyword-pattern:    keyword identifier {keyword identifier}.
temporaries:        "|" {identifier} "|".
```

Note that the pipes in the `temporaries` must be surrounded by spaces, and may not appear in column 1.

2.5.4 Include

With a program containing many classes, it is best to put each class into its own source file. This helps in readability, because you don't have to hunt through a file looking for one of several classes.

Such a program should include a "configuration file," which lists the files to be loaded. Each line of the configuration file will specify one file to be loaded.

```
include:              "Include" file-name.
file-name:            string
```

When Little Smalltalk reads an `include`, it reads the contents of the specified file, and then continues with the line after the `Include`.

Example

```
Include 'airline.st'
```

Chapter 3

The Class Library

3.1 Introduction to the class library

Little Smalltalk, as a language, does not include most of the capabilities which we would expect in any programming language. For example, the syntax does not define a way of even adding two numbers together.

Fortunately, the Little Smalltalk language includes an extensive *class library*, which defines classes representing the basic objects such as characters, numbers, and strings, and collections of objects, such as arrays, lists, sets, and dictionaries.

This chapter describes the class library. For each class, the most important messages to which instances of that class will respond have been shown, as well as a number of examples. The examples have all been run on Little Smalltalk; the dialogue with the system was cut and pasted into this document. Some examples have been reformatted to improve clarity (such reformatting has been noted in the example).

In studying this chapter, you should refer to Figure 3.1, which shows the hierarchy of classes in the standard library.

3.2 `Block`

A block is basically a procedure. Blocks are written with square brackets, with the argument names at the beginning:

```
>   A <- [:x :y | x print. y print.
                 'Th-th-that''s all, folks!' print. nil]
Block
>   A value: 3 value: 4
3
4
Th-th-that's all, folks!
nil
```

(This example has been reformatted for clarity.)

```
Object
   Block
   Boolean
      False
      True
   Class
   Collection
      IndexedCollection
         Array
            ByteArray
               String
         Dictionary
      Interval
      List
         Set
   Context
   File
   Link
   Magnitude
      Char
      Number
         Float
         Fraction
         Integer
            LongInteger
   Method
   Random
   Smalltalk
   Switch
   Symbol
   UndefinedObject
```

Figure 3.1: The Little Smalltalk class hierarchy

In this example, x and y are the names of the arguments, and the "pipe" symbol, |, is used to separate the arguments from the statements of the block. (The colons on the argument names are necessary, in order to keep the compiler happy.)

Blocks need not have arguments:

```
>   B <- ['The cat came back!' print. 43]
Block
>   B value
The cat came back!
43
```

- value
- value: x
- value: x value: y
- value: x value: y value: z

These messages cause the receiver to evaluate the block. The `value` message is used for blocks with no arguments; the other messages are used to specify values for blocks with 1, 2, or 3 arguments, respectively. (If you need more than 3 arguments for a block, you've probably designed your program wrong.)

- valueWith: anArray

This message is used if you want to evaluate a block with more than 3 arguments, or in some other strange cases. In most cases, you won't need it.

```
>   A <- [:a :b | ('>>>', a) print. ('<<<', b) print. 87]
Block
>   A value: 'foo' value: 'egg'
>>>foo
<<<egg
87
>   A valueWith: #('frodo' 'bilbo')
>>>frodo
<<<bilbo
87
```

- whileFalse: aBlock
- whileFalse

These messages allow you to create loops. The receiver evaluates itself, expecting a boolean result. If that result is `false`, then the receiver evaluates `aBlock`, and then repeats the process. If the result of the first block was `true`, then the loop terminates.

```
>   A <- 1
1
>   [A print. A <- A+1. A > 5] whileFalse:
       ['Looping' print]
1
Looping
2
Looping
3
Looping
4
Looping
5
Block
```

(This example has been reformatted for clarity.)

`whileFalse` is equivalent to `whileFalse: nil`, i.e., all of the work of the loop is done in the block which is the receiver.

- `whileTrue: aBlock`
- `whileTrue`

These messages allow you to create loops. The receiver evaluates itself, expecting a boolean result. If that result is `true`, then the receiver evaluates `aBlock`, and then repeats the process. If the result of the first block was `false`, then the loop terminates.

```
>   A <- 1
1
>   [A print. A <- A+1. A <= 5] whileTrue:
       ['Looping' print]
1
Looping
2
Looping
3
Looping
4
Looping
5
Block
```

(This example has been reformatted for clarity.)

`whileTrue` is equivalent to `whileTrue: nil`, i.e., all of the work of the loop is done in the block which is the receiver.

3.3 Boolean

There are two subclasses, `False` and `True`, each of which has exactly one instance, `false` and `true` respectively. (The lower case names are the objects, whereas the capitalized names are the classes.)

You don't have to worry about this entire mechanism. Rather than describe the protocol for each of the two subclasses, we'll describe what happens when the various messages are sent to the Boolean objects.

Note: `true` and `false` are not predefined variable names. They are special Smalltalk syntax for the two Boolean objects. In practice, all that this means is that you can't assign a new value to these names (not that any sensible person would want to!).

- `and: aBlock`

  if this message is sent to `true`, `aBlock` will be evaluated and its result returned. If the message is sent to `false`, then the return value will be `false`. Note that the second argument is a block. This is done so that it need not be evaluated unless necessary.

```
>   (1 < 2) and: [3 < 4]
true
>   (1 < 2) and: [3 > 4]
false
>   (1 > 2) and: ['Help!!! I got it wrong' print. false]
false
>   (1 < 2) and: ['I did it right' print. true]
I did it right
true
```

- `ifTrue: aTrueBlock ifFalse: aFalseBlock`

  return the value computed by either evaluating aTrueBlock or aFalseBlock, according to the receiver. Sending this message to `true` causes `aTrueBlock` to be evaluated; sending this message to `false` causes `aFalseBlock` to be evaluated.

```
>   (1 < 2) ifTrue: ['OK' print] ifFalse: ['Wrong!' print]
OK
OK
>   (3 < 2) ifTrue: ['Wrong' print] ifFalse: ['OK' print]
OK
OK
```

- `ifFalse: aBlock`
- `ifTrue: aBlock`

equivalent to `ifTrue:ifFalse`, except that the omitted block is assumed to be `[nil]`.

```
>   (2 < 1) ifFalse: [7]
7
>   (2 > 1) ifFalse: [7]
nil
>   (1 > 2) ifTrue: [5]
nil
```

- `ifFalse: aFalseBlock ifTrue: aTrueBlock`
  equivalent to `ifTrue: aTrueBlock ifFalse: aFalseBlock`. This is just here for people who like to be different!

- `not`
  returns the logical negation of its receiver. If this message is sent to `true`, then the value returned is `false`, and vice versa.

- `or: aBlock`
  if this message is sent to `false`, `aBlock` will be evaluated and its result returned. If the message is sent to `true`, then the return value will be `true`. Note that the second argument is a block. This is done so that it need not be evaluated unless necessary.

```
>   (1 < 2) or: ['I won''t be evaluated' print]
true
>   (1 > 2) or: ['I will be evaluated' print. true]
I will be evaluated
true
```

- `xor: aBoolean`
  returns the "exclusive or" of the receiver and the argument, i.e., `true` if one or the other is `true`, and `false` if both are `true` or both are `false`. Note that, unlike `and:` and `or:`, `xor:` requires that its second argument be a Boolean value directly, rather than being a block which produces a Boolean value. (Both the receiver and the argument of `xor:` must be evaluated, regardless of the value of the receiver.)

```
>   (14 > 15) xor: (12 < 7)
false
>   (14 < 15) xor: (12 < 7)
true
```

3.4 Class

Classes are objects, too, though you probably don't want to think too much about this fact.[1] The only real message you'll ever send to a class is a request to *instantiate* it, i.e., to create a new object of that class.

- `new`
  returns a new object of that class, and sends that object the message `new`.

- `new: anInteger`
  This message is used for classes such as arrays, where you need to say how many elements the array is to hold. `anInteger` gives that value. This message returns a new object of that class, with the specified size.

3.5 Collection

The collection classes are of major importance in the system. A collection is, as its name suggests, a way of organizing a group of objects. A number of different collection classes are provided in the system.

- `do: aBlock`
  This is a "virtual method," in that it is actually implemented by each subclass of `Collection`. `do:` sends each item of the collection to `aBlock`.

```
>   X <- Set new
Set ( )
>   X add: 3. X add: 5. X add: 8. X add: 2
Set ( 2 8 5 3 )
>   X do: [:y | ('>>>', (y asString)) print]
>>>2
>>>8
>>>5
>>>3
Set ( 2 8 5 3 )
```

- `occurrencesOf: anItem`
  returns the number of times `anItem` occurs in the collection given by the receiver. An item which is not found in the receiver occurs 0 times.

[1]If classes are objects, and objects are members of classes, then classes must be members of classes. But what class would a class be a member of? And what class would that class be a member of? Bertrand Russell proved that such problems, in general, don't have answers. Fortunately for us, Smalltalk's class hierarchy can be formalized properly. However, the details are complex enough that they're not worth understanding. If you're interested in the solution to this problem, see Tim Budd's *A Little Smalltalk*, and, for the real scoop, Goldberg and Robson's *Smalltalk-80: The Language*.

- `size: anItem`
  gives back the number of items in the receiver.

  There are a number of messages to collections which define various sorts of mappings. The description of these messages is T.B.A.

3.5.1 IndexedCollection

The major thing that an indexed collection has is the ability to locate items given by an *index value*.

- `at: aValue`
  returns the item in a collection whose index value is given by `aValue`. If no such item exists, generates an error.

- `at: aValue ifAbsent: aBlock`
  returns the item in a collection whose index value is given by `aValue`. If no such item exists, `aBlock` is evaluated.

```
>   D <- Dictionary new
Dictionary ( )
>   D at: 3 put: 7
Dictionary ( 7 )
>   D at: 3
7
>   D at: 4 ifAbsent: ['aooga' print]
aooga
aooga
>   D at: 3 ifAbsent: ['aooga' print]
7
```

- `at: anItem put: aValue`
  stores `aValue` in position `anItem` in the `IndexedCollection`.

- `keys`
  returns a set of the keys (index values) which may be used with this `IndexedCollection`.

```
>   #(13 9 27) keys
Set ( 3 2 1 )
```

Array

Arrays are `IndexedCollections` in which the key values are the integers from `1` to `n`, where `n` is the size of the array.

You can write an array using a special syntax:

```
>   #(1 2)
Array ( 1 2 )
```

The messages understood by arrays are basically those understood by `IndexedCollection`.

String

Strings are sequences of characters, enclosed in single apostrophes. If you need an apostrophe inside a string, use two consecutive apostrophes:

```
>   'hello'
hello
>   '9 o''clock'
9 o'clock
```

Strings are mutable arrays: they respond to `at:` and `at:put:` messages:

```
>   A <- 'hello'
hello
>   A at: 3 put: $r
herlo
```

Because of this mutability, when you're going to mutate a copy of a string you must remember to copy the string explicitly:

```
> A <- 'hello'
hello
> B <- A
hello
> C <- A copy
hello
> A at: 3 put: $r
herlo
> A, ' ', B, ' ', C
herlo herlo hello
```

In this example, when we mutated `A`, we also mutated `B`, because `A` and `B` are the same object. `C`, on the other hand, was made by *copying* `A`, and is, therefore a different object. Mutating `A` will not affect `C`, and vice versa.

Note that the elements of a string are themselves characters.

- `, aString`
  returns a new string consisting of the receiver and `aString` catenated together.

```
>   A <- 'hello, '
hello,
>   B <- 'world!'
world!
>   A,B
hello, world!
```

- `< aString`
- `> aString`
- `= aString`
- `<= aString`
- `>= aString`
- `~= aString`

The full complement of relational operators is provided. Comparisons are done by comparing characters from left to right. If one string is longer than another, then the longer string is considered greater.

```
>   'foo' < 'bar'
false
>   'foo' >= 'bar'
true
>   'foo' <= 'food'
true
>   'foo' = 'FOO'
false
```

- `copyFrom: position1 to: position2`
  returns a substring of the receiver, consisting of those characters between `position1` and `position2`, inclusive.

```
>   'hello' copyFrom: 1 to: 4
hell
>   'discombobulation' copyFrom: 4 to: 8
combo
```

`Dictionary`

Dictionaries are also `IndexedCollections`, and therefore respond to `at:`, `at:put:`, and `at:ifAbsent:` messages.

3.5.2 Interval

`Intervals` are collections of numbers, produced by sending `to:` and `to:by:` messages to numbers:

```
>   1 to: 10
Interval ( 1 2 3 4 5 6 7 8 9 10 )
>   1 to: 10 by: 2
Interval ( 1 3 5 7 9 )
>   1 to: 5 by: 0.5
Interval ( 1 1.5 2 2.5 3 3.5 4 4.5 5 )
>   (1/2) to: (9/10) by: (7/33)
Interval ( 1/2 5/6 )
>   (1/2) to: (19/10) by: (7/33)
Interval ( 1/2 47/66 )
>   (1/2) to: (19/10) by: (7/33)
Interval ( 1/2 47/66 61/66 25/22 89/66 103/66 39/22 )
```

`Intervals` are immutable, and, since they are not `IndexedCollections`, do not respond to the `at:` and `at:put:` messages. An `Interval` is stored in an extremely compact way, even if the number of items is very large.

The primary use for an `Interval` is to serve as the receiver for a `do:` message:

```
>   (1 to: 3) do: [:i |
      ((i asString), ' ', ((i*i) asString)) print]
1 1
2 4
3 9
Interval ( 1 2 3 )
```

(This example has been reformatted for clarity.)

3.5.3 `List`

Lists are singly-linked data structures, similar to those in other languages such as Lisp or Scheme. Each element of a list contains a value and a "next", which is the next element of the list.

Elements may be added at the beginning and the end of a list, or according to an "ordering block." Elements may also be removed from a list, either by removing the first element or an element with a specific value.

Since `List` is a subclass of `Collection`, the standard collection messages, e.g., `do:`, are supported.

- `add: aValue`
  Adds `aValue` at the end of the list.

- `add: aValue ordered: aBlock`
  Adds `aValue` as a new item in the list, positioned before the first item for which `aBlock` evaluates to `false`. `aBlock` is called with two arguments: the first is the value of the `List` element, and the second is the value being added.

- `addAll: aValue`
  `aValue` is a collection. Each of the elements of this collection is added, in an unspecified order, to the `List`.

- `addFirst: aValue`
  Adds `aValue` at the beginning of the list.

- `addLast: aValue`
  Same as `add:`.

- `do: aBlock`
  As with all collections, the `do:` message enumerates the `List`, i.e., passes each value of the `List`, in order from first to last, to `aBlock`.

- `first`
  Returns the first element of the `List`, without altering anything.

- `reverseDo: aBlock`
  Enumerates the `List`, but in reverse, i.e., goes from last to first.
  Example

```
>  x <- List new
List ( )
>  x add: 2
List ( 2 )
>  x add: 5
List ( 2 5 )
>  orderBlock <- [:a :b | a < b]
Block
```

```
>  x add: 3 ordered: orderBlock
List ( 2 3 5 )
>  x add: 17 ordered: orderBlock
List ( 2 3 5 17 )
>  x add: 1 ordered: orderBlock
List ( 1 2 3 5 17 )
>  x addFirst: 10
List ( 10 1 2 3 5 17 )
>  x addLast: 10
List ( 10 1 2 3 5 17 10 )
>  x removeFirst
List ( 1 2 3 5 17 10 )
>  x remove: 5
List ( 1 2 3 17 10 )
>  x first
1
>  x do: [:a | ('>>', a) print]
>>1
>>2
>>3
>>17
>>10
List ( 1 2 3 17 10 )
>  x removeFirst
List ( 2 3 17 10 )
>  x removeFirst
List ( 3 17 10 )
>  x removeFirst
List ( 17 10 )
>  x removeFirst
List ( 10 )
>  x removeFirst
List ( )
>  x == nil
false
>  x addAll: #(1 2 45)
List ( 1 2 45 )
```

`Set`

T.B.A.

3.6 Context

This class is used by the Smalltalk evaluator, and is not necessary for you to understand.

3.7 `File`

The `File` class allows you to work with text files stored on disk. The normal process is to *open* a file, which establishes a connection between a file on disk and a `File` object; then to *read* or *write* the file, as appropriate; and then to *close* the file, indicating that you no longer need access to it.

Rather than showing you individual examples, there is a set of `File` examples at the end of this section.

- `getString`
  tells the receiver to read a line (up to the newline character) from the disk. This only makes sense if the file was opened in *read* mode. If there is no more information to be read, the returned value will be `nil`. Otherwise, it will be the line of information read.

- `close`
  tells the receiver to close the file on disk. The effect of `close` is to ensure that everything you've asked to write to the file has been successfully written on the disk. If your program crashes before closing a file, things which you've written to the file via `print:` or `printNoReturn:` may or may not actually appear in the file.

- `name: aString`
  tells the receiver that the file on disk will be named `aString`.

- `open: aString`
  `aString` is a so-called *file mode*, which says what you're doing with the file. Three modes are valid:

  - `'r'` says you're going to *read* the file (transfer information from disk into the computer). The file must already exist on disk before you attempt to open a file with this mode.
  - `'w'` says you're going to *write* the file (transfer information from the computer to disk). The file itself may or may not exist on disk beforehand. If so, its contents will be erased.
  - `'a'` says you're going to *write* the file (transfer information from the computer to disk), but retain the existing contents of the file; what you write will be *appended* to the existing contents.

- `print: aString`
- `printNoReturn: aString`
  These messages write `aString` to the file. The only difference is that `print:` puts a newline after what's written, but `printNoReturn:` doesn't.

3.7.1 File examples

Create a file, by writing something to disk:

```
>   F <- File new name: 'junk.txt'
File
>   F open: 'w'
File
>   F print: 'I am a file'
File
>   F printNoReturn: 'Hello,'
File
>   F print: ' world!'
File
>   F print: 'Bye for now!'
File
>   F close
File
```

This creates the file `junk.txt`, with the following contents:

```
I am a file
Hello, world!
Bye for now!
```

To read this file back, we might write:

```
>   F <- File new name: 'junk.txt'
File
>   F open: 'r'
File
>   [tmp <- F getString. tmp ~~ nil] whileTrue:
        [('>>>',tmp) print]
>>>I am a file

>>>Hello, world!

>>>Bye for now!

Block
```

Note that the output appears to be double-spaced. This is because the input lines read from the file have a newline character at the end, and `print` puts a second newline on, itself.

3.8 Magnitude

A magnitude is something which can be compared. Characters and numbers are subclasses of class `Magnitude`.

- `< aMagnitude`
- `> aMagnitude`
- `= aMagnitude`
- `<= aMagnitude`
- `>= aMagnitude`
- `~= aMagnitude`

These are the six relational operators, which are defined on magnitudes, and hence on numbers and characters. They return true if the relation is satisfied and false otherwise.

```
>   2 <= 3
true
>   2 > 3
false
>   2 = 3
false
>   2 ~= 3
true
>   $a < $b
true
>   $a > $b
false
```

3.8.1 Character

Characters are written with a preceding dollar sign, thus:

```
$a  $$  $ 
```

The third example is the character “space.”

- `< aChar`
- `> aChar`
- `= aChar`
- `<= aChar`
- `>= aChar`
- `~= aChar`

< returns true if receiver is less than aChar, using the ASCII code. The other comparison messages are similar.

- `asInteger`
  returns the numeric character code corresponding to the receiver.

- `asString`
  returns a string of length 1 corresponding to the receiver. This is particularly useful if you want to make a string with unprintable characters in it:

```
BellChar <- 7 asChar.
('Please enter YES or NO', (BellChar asString)) print.
" On many terminals, sending the character with code 7
  rings the bell."
```

- `isAlphabetic`
- `isBlank`
- `isDigit`
- `isLowercase`
- `isUppercase`
  return true if the receiver satisfies the condition, and false otherwise.

```
> $A isLowercase
false
> $7 isDigit
true
```

3.8.2 Number

There is actually a fairly large set of number classes. They are all subclasses of `Number`:

- `Integer` and `LongInteger`, which between them support positive and negative numbers of effectively unbounded range. In most cases, you don't have to worry about the difference between the two classes. **Bug:** sometimes the system uses `Float` where it should use `LongInteger`.

- `Fraction`.

- `Float`, a decimal number, so-called because the decimal point can "float" anywhere within the number.

Numbers are written in standard forms, e.g.,

```
1 -3 2/3 -0.625 3.5e2 3.5e-2
```

(`3.5e2` is another way of writing `350`; `3.5e-2` is another way of writing `.035`).

Note that `2/3` is an expression, and therefore you must put it in parentheses if you're using it in a complex expression:

```
>   2/3 + 1/3
5/9
>   (2/3) + (1/3)
1
```

For the most part, you don't have to keep these classes separate in your own mind; if you combine numbers from different classes, things will work properly.

```
>   2.3 + 4
6.3
>   2.3 + (4/7)
2.871429
>   1 + (9/11)
20/11
>   (4/7) + 2.3
2.871429
>   (1/3)+(2/3)
1
```

As you can see, the number classes are organized in such a way that arithmetic messages give back a result which is in the form of the most "general" argument (integers are the least general, then fractions, and finally floats). With fractions, if the result has denominator `1`, it will be converted to an integer.

- `print`
  (Overrides `Object` method.) Displays the numeric representation of the receiver on the screen, and returns `self`.
  Bug: Some long integers print out with incorrect leading zeroes; the value printed is correct, though.

- `+ aNumber`
- `- aNumber`

- `* aNumber`
  return the result of addition, subtraction, or multiplication, respectively.

- `/ aNumber`
  if both the receiver and `aNumber` are integers or fractions, returns a fractional result. Otherwise, returns a float result.

```
>   2/3
2/3
>   2.0/3
0.666667
>   2/3.0
0.666667
>  (1/6) / (1/3)
1/2
>   (1/4) / (1/3)
3/4
```

- `abs`
  returns the absolute value of the receiver.

- `asFloat`
  returns the receiver (either an integer or a fraction) as a float.

```
>   (2/3) asFloat
0.666667
```

- `bitAnd: anInteger`
- `bitInvert`
- `bitOr: anInteger`
- `bitXor: anInteger`
- `bitShift: anInteger`

These messages provide "bitwise" logical operations on Integers. Each bit of the binary representation of an integer is taken as a Boolean value (with `1` meaning true and `0`—meaning false), and combined according to the standard definition of "and," "[inclusive] or," and "exclusive or." `bitInvert` returns a result in which every `1` in the binary representation of the receiver has been replaced by a `0`, and vice versa. `bitShift:` gives back an integer which has been multiplied or divided by $2^{\texttt{anInteger}}$, depending upon whether `anInteger` is positive or negative, respectively.

```
>   126 bitAnd: 15
14
>   17 bitOr: 26
27
>   15 bitInvert
-16
>   5 bitShift: 2
20
>   18 bitShift: -1
9
```

In this example, note that $126_{10} = 01111110_2$, and $15_{10} = 00001111_2$. Combining these following the `and` rule gives us $00001110_2 = 14_{10}$. Note that $-16_{10} = 11110000_2$, which is the correct result for `15 bitInvert`.

Note that these messages only work for `Integers`, which have values between ± 16383. Attempts to go beyond that range will produce errors.

- `quo: aNumber`
- `rem: aNumber`

returns the quotient or remainder computed by dividing the receiver by `aNumber`. Note that these messages are only defined for integers or floats, but not fractions. Also note that the remainder function is not clearly defined for negative divisors and/or dividends. You will get an answer, but it might vary from machine to machine, and it may or may not be what you expect.

- `timesRepeat: aBlock`

This message may only be sent to integers. It commands that the integer repeat the block the number of times given by the receiver's value.

```
>   3 timesRepeat:
      ['This is the way the world ends...' print]
This is the way the world ends...
This is the way the world ends...
This is the way the world ends...
```

(This example has been reformatted for clarity.)

- `to: aNumber`
- `to: aNumber by: aNumber`

return an `Interval` as specified by the arguments.

```
>   1 to: 10
Interval ( 1 2 3 4 5 6 7 8 9 10 )
>   1 to: 10 by: 2
```

```
Interval ( 1 3 5 7 9 )
>   0.7 to: 1.5 by: 0.3
Interval ( 0.7 1 1.3 )
>   1/2 to: 3/2 by: 1/4
Interval ( 1/2 3/4 1 5/4 3/2 )
```

- `truncated`
- `rounded`
- `ceiling`
- `floor`

returns the result of converting the receiver to an integer. `truncated` rounds towards zero; `round` rounds towards the nearest integer; `floor` rounds towards negative infinity; and `ceiling` rounds towards positive infinity.

Bug: these operations don't produce the correct result when the receiver is negative.

3.9 Method

Objects of this class represent the methods for a class. You don't need to worry about this class.

3.10 Object

This is the root class in the entire system. Every other class inherits from it, either directly or indirectly.

- `== aValue`

returns `true` if the receiver and `aValue` are the same object (*not* two different objects with the same value), and `false` otherwise.

```
>   A <- #(1 2 3)
Array ( 1 2 3 )
>   B <- #(1 2 3)
Array ( 1 2 3 )
>   A = B
true
>   A == B
false
```

In this example, we created two different arrays, which happened to have the same value. `=` determines whether the contents of two objects are equal, which they were in this example; `==` determines whether the receiver and the argument were the same object (i.e., take up the same memory locations), which they were not, because they were created separately.[2]

[2]The distinction here is exactly the same as that between `equal` and `eq` in Scheme or Lisp.

- `~~aValue`

returns `true` if the receiver and the argument are not the same object, and `false` otherwise. `~~` therefore returns the negation of `==`.

- `asString`

returns a string representation of the receiver. The `asString` method in `Object` just returns the string name of the class of the object, but this method is often overridden in subclasses, e.g., numbers, characters, and arrays.

- `copy`
- `deepcopy`
- `shallowCopy`

returns a copy of the receiver. `shallowCopy` and `deepCopy` differ in that `deepCopy` recursively copies all of the objects within the receiver, while `shallowCopy` only copies the receiver itself. `copy` is defined to do either a shallow or a deep copy, depending upon what is appropriate for the receiver.

- `hash`

returns a "hash code" for the object. This will be a number which often appears to have no independent significance, but sending `hash` to the same object a number of times will give the same hash value back each time.

```
>   3 hash
3
>   'The world is funny' hash
1717
>   true hash
6042
>   true hash
6042
```

- `isNumber`

returns true if sent to any kind of number, false otherwise.

- `isChar`

returns true if sent to a character, false otherwise.

- `isKindOf: aClass`
- `isMemberOf: aClass`

these return `true` if the receiver is a member of (or is a kind of) the specified class, and false otherwise. The difference between membership and kindness [sic!] is that to be a member of a class, an object must not be a member of a subclass of that class. This message is extremely useful for error checking.

```
>   3 isMemberOf: Integer
true
>   3 isKindOf: Integer
true
>   3 isMemberOf: Number
false
>   3 isKindOf: Number
true
```

- new

This is the default object initialization method. When an object is created (by sending the message `new` to a Class) , the first thing is to send a `new` message to the newly-created object. The `new` method in `Object` does nothing (i.e., it just returns `self`), but many subclasses override this default method).

Note a particularly insidious bug:

```
>   A <- Directionary new
nil
>   A class
UndefinedObject
```

In this example, the user meant to instantiate `Dictionary`, but made a spelling error. The result was that `Directionary` evaluated to `nil`. It's not an error to send `new` to `nil`: class `UndefinedObject` (of which `nil` is the sole instance) inherits the default `new` method from `Object`. The result is that instead of `A` being a `Dictionary`, it's just plain old `nil`.

- print

This is the default message to send a representation of an object to the screen. The `print` method in `Object` just displays the name of the class, but many subclasses override this message to produce a more reasonable form of output.

```
>   [3 + 4] print
Block
Block
>   'hello' print
hello
hello
```

In this example, note that `Blocks` have no `print` method, and therefore inherit `Object`'s; `Strings`, on the other hand, override this method so as to print the contents of the string, rather than just `String`.

- `respondsTo: aSymbol`
  returns `true` if the receiver will accept the message named `aSymbol`, and false otherwise. This is useful for "sounding out" an object, to see whether it will respond to that message, before sending it. (Sending an invalid message will, of course, result in a Smalltalk error.)

```
>   A <- Dictionary new
Dictionary ( )
>   A respondsTo: #new
true
>   A respondsTo: #at
false
>   A respondsTo: #at:
true
>   A respondsTo: #at:put:
true
```

Note that in the case of multiple keyword messages, the message name is the concatenation of the individual keywords, with colons preserved.

3.11 Random

`Randoms` are random number generators. An instance of `Random` will return numbers chosen uniformly from a range given by the arguments to the various messages.

I cannot say how good the random number generation algorithm is. Don't depend upon it being wonderful.

Each `Random` has as an instance variable a "seed," which is an integer which is used to generate the next random number. Setting the seed to a defined value allows you to control the random numbers produced.

- `between: low and: high`
  returns the next random number, as a fraction, between `low` and `high` inclusive.

```
>   R <- Random new
Random
>   5 timesRepeat: [(R between: 0 and: 5) print]
17/50
329/100
999/200
```

```
163/200
27/25
5
```

The last value, `5`, is not an output from the random number generator. It's just the value of the whole `timesRepeat:`.

- `next`
  Give back the next random number, as a fraction in the range from 0.0 to 1.0. (1.0 itself will never be returned.)

```
>   5 timesRepeat: [(R next asFloat) print]
0.698
0.465
0.78
0.095
0.462
5
```

The last value, `5`, is not an output from the random number generator. It's just the value of the whole `timesRepeat:`.

- `next: anInteger`
  returns a List of `anInteger` random fractions between 0.0 and 1.0. (1.0 is not returned).

```
>   R <- Random new
Random
>   R next: 10
List ( 403/500 777/1000 86/125 67/500 1/40 18/25 839/1000
       907/1000 3/125 63/500 )
```

3.12 Smalltalk

This is another one of those classes which keeps the system running. In general, you don't need to worry about it.

There is one instance of this class, named `smalltalk` (note lower case), which the system creates when it starts up. You can use this object to produce error messages.

- `error: aString`
  generates a run-time error message, and halts your program.

```
> smalltalk error: 'I feel tired'
I feel tired
Process trace ( Process)
Smalltalk error: ( Smalltalk String)
Block value ( Block)
Method executeWith: ( Method Array)
String execute ( String)
String value ( String)
```

3.13 Switch

This is another one of those classes which keeps the system running. You don't need to worry about it.

3.14 Symbol

This class implements "atomic" symbols. A symbol is written with a preceding `\#`, e.g., `#foo`, `#+`. Symbols have no real behavior (though they can respond to the various messages which can be sent to objects, such as `asString`).

3.15 UndefinedObject

This class has only one instance, the object written as `nil`. This object is the value of all variables which have not been assigned any other value.

Note that the only way you can test something to see whether it's nil is to use `==` or `~~`. The arithmetic/string/character comparison messages, such as `=`, are not defined for this class.

Chapter 4

Errors

Smalltalk errors fall into two main classes: compiler errors and run-time errors. This chapter discusses the symptoms and treatment of these errors.

4.1 Compilation errors

When a Smalltalk file is loaded in, or when you type in an expression, the string containing your code is presented to the *compiler*, one of the modules of the system. The compiler analyzes this string, and prepares an array of *bytecodes* (machine instructions) which correspond to it. That array is what gets executed when the code is executed.

At present, the messages produced by the compiler are fairly random. Some errors in a file will inhibit loading of the file, and actually terminate execution. Others will produce a message which may or may not be informative. Since I hope to modify the compiler fairly extensively in order to improve the messages, I will not bother to describe the present compiler time error handling in more detail here.

However, there are some things you should watch out for when an error is detected. Omitted colons are a source of trouble, as are misuses of the pipe symbol `|`. The pipe symbol is used in three different ways in Little Smalltalk:

- as a method separator, if it appears in the leftmost column of the line;
- as a temporary variable flag, if it appears as the first symbol (excluding comments) of a method, and is surrounded by spaces; and
- as a binary message which can be sent to an object.

Here's a (nonsensical) example of the use of the pipe symbol for all three purposes:

```
   ...
   ^ x + y   "End of the previous method."
|
   infundibulate: x
   | temp |
   temp <- x | 3.
   ...
```

Note that if we'd written the declaration of `temp` as `|temp|`, we'd have gotten some strange error messages.

4.2 Run-time errors

Most run-time errors, such as division by zero, cause an `error:` message to be sent to `smalltalk`. This, in turn, halts the program and generates a "walkback:"

```
> 3 / 0
division by zero
Process trace ( Process)
Smalltalk error: ( Smalltalk String)
Integer / ( Integer Integer)
Block value ( Block)
Method executeWith: ( Method Array)
String execute ( String)
String value ( String)
```

In this walkback, the first line is a description of the error. Successive lines show the methods which are currently active. In this example, ignore the line `Process trace (Process)`; the line below that says that we were executing the `error:` method of class `Smalltalk`. In turn, that method was executed from the `/` method of class `Integer`. That indeed was the problem. (The lines below that in the walkback correspond to various methods which are used to read in and execute code from the user.)

Let's take another example. Suppose we create the following file:

```
   Class Demo Object foo
   Methods Demo 'all'
   [
      new
         fooo <- 0
   |
      count
         foo <- foo+1
   ]
```

When we run this, the following "disaster" occurs:

```
> a <- Demo new
Demo
> a count
not recognized UndefinedObject #+
```

```
Process trace ( Process)
Smalltalk error: ( Smalltalk String)
Object message:notRecognizedWithArguments:
    ( UndefinedObject Symbol Array)
Demo count ( Demo)
Block value ( Block)
Method executeWith: ( Method Array)
String execute ( String)
String value ( String)
```

(This example has been reformatted for clarity.)

The operative information here is on the top line: we tried to send the `+` message to an `UndefinedObject`. This is, needless to say, a BAD THING.

Where in the program did this occur? Scanning down the walkback, we see that we were executing methods from classes `Process`, `Smalltalk`, and `Object`, system classes all. But the next line is a gem: we were executing `Demo`'s `count` method. Take a look at the code there; you will find that we're trying to compute `foo+1`. Clearly `foo`'s value is undefined (if you weren't sure, you could always stick in a `foo print` debugging statement at the beginning of the method).

But why is `foo` undefined? We initialized it in the `new` method, which is executed when we instantiate `Demo` (i.e., send the class the `new` message). Aha! What we really did was to initialize `fooo`, an anonymous global variable.[1]

In summary, the walkback information is vital to tracking down problems: the first line tells you what's wrong, and the subsequent lines tell you which methods are currently active. The parenthesized names are the classes of the receiver and the arguments. For example, examine `Number / ( Float Integer)`, which is one of the walkback lines when you try to compute `4.5/0`. This says that we're trying to execute the `/` method in class `Number`. The receiver of the message is an object of class `Float`; the argument to the `/` message is an `Integer`. Why isn't the receiver's class the same as the class in which the method is found? Because, in this case, `Float` is a subclass of `Number` and is relying upon an inherited `/` method.

[1]It is a regrettable deficiency of the present compiler that no warning about this error is presently generated. That will be fixed later.

Method Index