1

The Squeak Syntax Reference

In this chapter I would like to introduce Squeak's syntax to programmers of other languages. Note that Squeak is a Smalltalk, so any book on Smalltalk can be used to obtain another point of view on the syntax and the model. Smalltalk was influenced by the following languages: Lisp (lexical closure, uniformity, control constructs defined in the language itself), LOGO (a dynamic functional language), and Simula (object-oriented programming).

The syntax of Squeak is based on (1) 6 reserved identifiers; (2) constant expressions called literal objects, which includes numbers, characters, strings, symbols, and arrays; (3) identifiers and assignment; (4) messages; and (5) block closures (an anonymous and deferred sequence of expressions). Based on these syntactical elements, conditional expressions and iterations are naturally expressed without requiring any dedicated constructs. Smalltalk is a dynamically typed language, and therefore you never have to specify the type of a variable or method argument.

Parts of this chapter are freely adapted from the *Squeak Quick Reference Manual*, written by Andrew C. Greenberg and Andrew P. Black. I thank them for the work they have done and for granting me permission to make use of it. I will begin by giving you an overview of the syntax and will then go step by step into the details. If you are interested only in the syntax, you skip the first section, which begins with some examples. I will illustrate the syntax using some small scripts, and I suggest that you try out all the expressions contained therein.

1 Uniformity and Simplicity

As you will soon discover, the Smalltalk syntax does not introduce specific constructs for conditionals, loops, class definition, or method definition. All of these are expressed using the basic syntactical elements, that is, messages and blocks. Hence a condition is a message sent to a Boolean (see Section 13.1), a loop is a message sent to an integer or a collection with the code that is to be repeated expressed as a block-closure (see Section 13.3 13.4).

The syntax of Smalltalk is simple if one can manage to get rid of old reflexes created by other languages such as C, C++, and Java and believe that a simpler alternative is possible. You should realize that originally, the intention of Smalltalk's inventors was to develop a language that children could master. Therefore, the code looks like English sentences and can be more or less read aloud.

Ralph Johnson, one of the four authors of the Design Patterns book, frequently says that the syntax of Smalltalk can be fit on a postcard, as shown by method 1.1. Obviously, this method does not do anything useful, but it really does contain all the syntactical elements of Smalltalk!

Method 1.1

```
exampleWithNumber: x
"This is a small method that illustrates every aspect of the Smalltalk method syntax
except primitives, which aren't very standard. It has unary, binary, and keyword
messages; declares arguments and temporaries (but not block temporaries); accesses a
global variable (but not class and instance variables); uses literals (array, character,
symbol, string, integer, float); uses the pseudovariables true, false, nil, self, and
super, and has sequence, assignment, return and cascade. It has both zero-
argument and one-argument blocks. However, it doesn't do anything useful."

  |y|
  true & false not & (nil isNil) ifFalse: [self halt].
  y := self size + super size.
  #($a #a 'a' 1 1.0)
    do: [:each | Transcript
                show: (each class name);
                show: (each printString);
                show: ' '].
  ^ x < y
```

1.1 Syntactic Elements

Here is a summary of the syntactical elements of Squeak. Details will come later.

- $ defines a character.
- ' ' character string. Use two single quotes to represent an apostrophe or single quote inside the string: 'here', 'it"s fun'
- # defines a symbol (uniquely identified character strings). #IAmUnique
- #( ) defines an array at compile time. #(a b #(true))
- {} defines an array at run time. Thus {1+3 . 2*5} defines an array with elements 4 and 10.
- <primitive:...> defines a virtual machine primitive call.
- " " comment: "this is a comment"
- By convention, local or private variables start with a lowercase letter, and global variables start with an uppercase letter. Hence class names start with an uppercase letter because they are global variables. Thus, true is an instance of the class True.
- | | declaration of local variables in a context, method, or block-closure.
- := (and _, which appears as an arrow (← in Squeak), allows the assignment of the contents of a variable.

Examples 1.1

```
| x y |
x := 3.
y := x + 1.
y
```

- self, super, nil, true, false, and thisContext are reserved identifiers, also called *pseudovariables* because they are read-only and their values are defined by the system.
- ^ returns a result from a method or a block-closure. ^ acts as an escape mechanism. It escapes the method that contains it even if deeply nested in a block-closure or conditional. By default, every method returns the receiver (self), but the code browser will show the receiver only if necessary.
- . is an expression separator (and not a terminator, and thus not required at the end of a script).
- ; (called *cascade*) makes it possible to send several messages to the same object.
- [ ] defines a block-closure (lexical closure), which is a first-class object as in Lisp.
- :x defines an argument of a block-closure.
- | end of argument declaration in a block-closure.

Examples 1.2

```
[:y| y + 2] value: 3 "returns 5"
```

1.2 Two Graphical Examples

Before going into the details of the syntax, I will present two scripts.

Image Capture. Squeak supports bitmap manipulation (classes BitBlt and WarpBlt) as well as the usual image formats such as GIF, JPEG, and PNG. Script 1.1 captures a part of the screen and saves it in a file.

Script 1.1 (*Capturing an image)*

```
| im |
im := Form fromUser.
GIFReadWriter putForm: im onFileNamed: 'test.gif'
```

The script declares the temporary variable im, asks the user to capture a part of the screen by sending the unary message fromUser to the class Form, and then saves the captured part as a GIF file by sending the keyword-based message putForm:onFileNamed: to the class GIFReadWriter. Note that the method name is really putForm:onFileNamed:, that is, including the two colons.

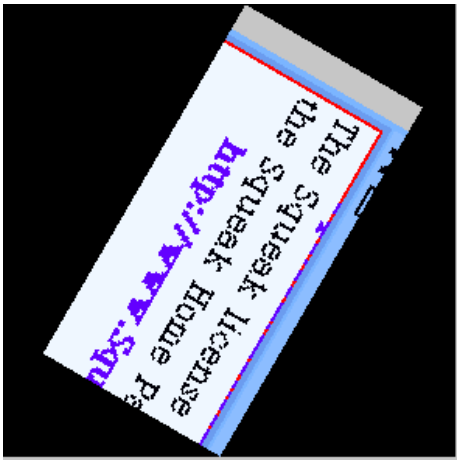

Figure 1.1: Rotating and zooming a part of the screen.

Image Transformations. All image formats are in fact internally transformed into instances of the class Form. A Form allows multiple transformations such as flip, rotation, and zoom. Script 1.2 describes how to rotate and zoom the portion of screen where the mouse is located (as illustrated in Figure 1.1).

Script 1.2 (*Playing with images)*

```
| angle form rotatingForm |
form := Form fromDisplay: (0@0 extent: 300@300).
angle := 0.
[Sensor anyButtonPressed] whileFalse: [
  rotatingForm := (Form fromDisplay:
          (Rectangle
              center: Sensor cursorPoint
              extent: 130@66)).
  (rotatingForm
    rotateBy: angle
    magnify: 2
    smoothing: 1) display.
  angle := angle + 5].
form display
```

Let's look at what is going on in this script.

- First of all, after having declared three variables (angle, form, and rotatedForm), we capture and store in the variable form a part of the screen by specifying a rectangular area. The idea is that we will restore the display when we have finished our manipulation. (0@0 extent: 300@300) returns a rectangle, and @ is a message invoked with a number, which returns a point.
- Then we initialize the variable angle. Then, as long as no button is pressed, we create a new Form from a screen area based on the position of the mouse and store it in the variable rotatedForm. Sensor cursorPoint returns the position of the mouse. Rectangle center: Sensor cursorPoint extent: 130@66 returns a rectangle centered on the mouse position of size 130 by 66 pixels.

- Then we ask this form to rotate about itself while we enlarge it. Note that the name of the method is rotateBy:magnify:smoothing:, and thus it has three arguments. Once it has been rotated, we ask the form to display itself, and then we increment the angle. We continue until a key is pressed.
- When a key is pressed, we redisplay the original part of the screen.

2 Pseudovariables: Smalltalk Reserved Identifiers

In Smalltalk, there are six reserved identifiers, nil, true, false, self, super, and thisContext, called pseudovariables because they are predefined and read-only. The pseudovariables true, false, and nil are constants, while self, super, and thisContext vary dynamically as code is executed.

- self is the Smalltalk equivalent of this. It always refers to the receiver of the message. super also refers to the receiver of the messages but changes the method lookup so that it starts from the superclass of the class containing the method that issues it.
- true, false represent Boolean values. They are the unique instances respectively of the classes True and False. And yes, everything is an object, even the Boolean values. Note, however, that the compiler inlines many Boolean invocations at the byte-code level.
- nil is the equivalent of NULL or null and refers to an undefined value or no value. nil is the unique instance of the class UndefinedObject. nil is the default value of instance variables.
- thisContext is a variable that represents the current execution stack. Note that such a stack is created only when the variable is accessed for obvious performance reasons, but it is quite useful for building development tools.

3 Comments and Identifiers

Comments. A comments is any sequence of characters delimited by " and ". Repeat the double quotes to include them in a comment.

Examples 1.3

```
"a comment:  any sequence of characters, surrounded by double quotes"
"comments can include the 'string delimiting' character"
"and comments can include double quote characters by ""doubling"" them"
"comments can span many
many
lines"
```

Identifiers. An identifiers is any sequence of characters that begins with an alphabetical character. This definition may vary, so avoid to use any cryptic nonalphabetical character if you want your code to look like Smalltalk code.

- Smalltalk identifiers are case-sensitive. Hence lunchTime and LunchTime are different identifiers.
- It is a Smalltalk convention for identifiers (instance variables and temporary variables) of several words to begin with a lowercase character, and then each subsequent word is capitalized (e.g., thisIsAnIdentifier).

- Certain identifiers, for example, global variables such as Transcript, and class variables, are by convention initially capitalized. The names of all classes are also global variables (e.g., OrderedCollection).

4 Literals (Constant Expressions)

Constant expressions are expressions that are evaluated at compile time. In the following, *—Printing the returned value:* means "prints as."

4.1 Numbers

- Squeak supports small-integer arithmetic defined in the class SmallInteger with fast internal primitives.
- Squeak supports arbitrary-precision arithmetic seamlessly (automatically coercing instances of SmallInteger to instances of LargePositiveInteger and LargeNegativeInteger where appropriate), albeit at a slight cost in speed. Try (SmallInteger maxVal + 1) class. We add one to the largest small integer, and we get the smallest LargePositiveInteger. Try the following expression to see that Squeak can easily represent huge numbers: 1000 factorial or 1000 factorial / 999 factorial.
- Squeak supports several other kinds of numerical entities, such as fractions (Fraction, arbitrary-precision rational numbers), and points (Point). While there are no literals for these objects, they are naturally expressed as operations on built-in literals. (2/3 and 2@3, respectively). Try the following to see automatic coercion at work: 1000 factorial / 999 factorial or (9/11) + (2/11).
- Numbers may be represented in many bases (radices), but the radix specification itself is always expressed in base 10. The base for the exponent part is the same as the radix. Thus 2r1010 *—Printing the returned value:* 10, 10e2*—Printing the returned value:* 1000 ($= 10 * 10^2$), but 2r1010e2*—Printing the returned value:* 40 ($= 10 * 2^2$).

Examples 1.4

Decimal integer: 1234, 12345678901234567890
Octal integer: 8r177, 8r1777777777777777777777
Hex integer: 16rFF, 16r123456789ABCDEF012345
Arbitrary-base integer: 2r1010 ==> 10
Integer with exponent: 123e2 ==> 12300, 2r1010e2 ==> 40
Float (double precision): 3.14e-10
Arbitrary-base float: 2r1.1 ==> 1.5
Float with exponent: 2r1.1e2 ==> 6.0

4.2 Characters (Instances of Class Character)

Printable characters begin with the sign $. To obtain a unprintable character, such as tab, cr, or space, send a message to the class Character.

Examples 1.5

```
$x "A character is any character (even unprintable ones), preceded by a dollar sign"
$3 "Don't be shy about characters that are digits"
$< "or symbols"
$$ "or even the dollar sign"
Character space "To get character for space, carriage return, tab"
Character cr "send a message to the class itself"
Character tab
```

4.3 Constant Arrays (Instances of Class Array)

The #() construct defines compile-time arrays. #() are also called *constant arrays*. See Section 5 for run-time-created arrays.

- Constant arrays are constants, and their elements must therefore be constants. Expressions that could represent the array elements are not evaluated, but are parsed as sequences of symbols as in #(1 + 3), where we get an array with three elements and not an array with a single element representing the sum.
- Constant arrays may contain constant arrays. The hash sign for internal constant arrays is optional.
- Identifiers and sequences of characters in constant arrays are treated as symbols; the hash sign for internal symbols is optional.
- Arrays are indexed with the first element at index 1.

Squeak also includes a syntax for creating arrays at run time; see section 5 below.

Examples 1.6

```
#(1 2 3 4 5) "An array of size 5 comprising five Integers (1 to 5)"
#('this' #is $a #'constant' array)
"An array of size 5 comprising
 a String ('this'), a Symbol (#is), a Character ($a),
 and two Symbols (#constant and #array)."
#(1 2 (1 #(2) 3 ) 4)
"An array of size 4 comprising two Integers (1 and 2),
an Array of size 3, and another Integer (4)."
#(1 + 2)
"An array of size 3 comprising 1, #+, and 2.
It is not the singleton array comprising 3."
```

5 Braces

Literal arrays defined using the construct #() are constant: They are created when the method is compiled, and the values they contain at creation time can be only literal objects (string, numbers, symbols, Booleans, and arrays).

To create arrays having objects other than literals, we have to explicitly add them, as shown in Example 1.7, which is sometimes boring to type.

Examples 1.7

```
| ar |
ar := #(nil nil nil).
ar at: 2 put: (100 factorial)

"or"
| ar |
ar := Array with: nil with: (100 factorial) with: nil
```

In contrast to the other Smalltalk dialects, Squeak provides a way to define nonliteral arrays in a compact form using {, }. Arrays created using { } are called *brace arrays*. Brace arrays are computed at run time. The elements of a brace array are the results of the execution of the expressions that compose it. Periods are used to separate the expressions.

Examples 1.8

```
{ 1. 2 . 3 . 4 . 5 }
"An array of size 5 comprising five Integers (1 to 5)"

| blop |
blop := 100 factorial.
{ $a . #brace . blop + 1}
"An array of size 3 comprising a Character ($a), a Symbol (#brace),
 and the value of the variable blop plus one."

{ 1 + 2 } "An array of size 1 comprising the single integer 3."
```

5.1 Strings (Instances of Class String)

A string is a sequence of characters delimited by a single quote '. Repeating the single quote allows one to include a quote in a string. A string is an array containing characters. Accessing a string returns the character at the corresponding position, starting with 1. Try: 'bell' at: 1 and 'bell' at: 1 put: $w.

Examples 1.9

```
'a string comprises any sequence of characters, surrounded by single quotes'
'strings can include the "comment delimiting" character'
'and strings can include embedded single quote characters by doubling'' them'
'strings can contain embedded
newline characters'
'' "and don't forget the empty string"
$a asString  "the string 'a'"
String with: $a
```

5.2 Symbols (Instances of Class Symbol)

Symbols are *unique* strings. Symbols start with the character #.

- The class Symbol is a subclass of String, and understands, in large part, the same messages.

- The primary difference between a symbol and a string is that all symbols making up the same sequence of characters are the *same* instance. Two different string instances can both have the characters 'test one two three', but every symbol having the characters #'test one two three' is the *same* and *unique* instance. This *unique instance* property means that symbols can be efficiently compared, because equality (=) is the same as identity (==). Try: 'aaa' == 'aaa' and #aaa == #aaa.
- Symbols containing colons (e.g., #at:, #a:keyword:selector:) are often referred to as keyword selectors because they are used to represent method names.
- Single or dual symbol (e.g., #* or #++) are often referred to as binary selectors. The following are permissible special characters for binary symbols: +-/*~ <=>@%|&?!. Note that #– is not a symbol (or a binary selector). On the other hand, #'–' is a symbol (but not a binary selector).

Examples 1.10

```
#'A string preceded by a hash sign is a Symbol'
#orAnyIdentifierPrefixedWithAHashSign
#orAnIdentifierEndingWithAColon:
#or:several:identifiers:each:ending:with:a:colon:
#- "A symbol can also be a hash sign followed by '-' or any special character"
#+< "or a hash signe followed by any pair of special characters"
```

6 Temporary Variable Definition and Assignment

Temporary Variables. To define a temporary variable we surround it by |. | a | has no value.

Examples 1.11

```
| aTemporaryVariable | "a temporary variable"
| var1 var2 | "Two temporary variables"
```

Assignments. Variable assignments are expressed using the sign _, ←, or :=.

Examples 1.12

```
identifier _ expression
identifier <- expression
identifier :=  expression
" := is always a legal alternative to _ , but the pretty printer
 uses _ or <- "
```

- The identifier (whether instance variable, class variable, temporary variable, or otherwise) will thereafter refer to the object returned by the expression.
- The " ←" glyph can be typed in Squeak by keying the underscore character (shift-hyphen).
- Assignments are expressions; they return the result of the right-hand side expressions. Note that this is considered bad-style to write code that rely on that.

7 Messages

In Smalltalk, except for the syntactic elements listed in the preceding section (: = ^ . # () [ ] | :=), everything is a message send. But there is no concept of operator, nor of overload. Thus you can define the method + on a class of your choice (in contrast to Java), but you do not have the possibility of defining their precedence, as in C++. Moreover, you cannot overload a method. The simplicity of the model has a price.

> A message is always sent to a message receiver, which can be a single object or the result of another message or expression.

Smalltalk has three types of message sends: *unary*, *binary*, and *keyword-based*. This distinction is based on the way the method name is written.

Unary Messages. Unary messages are messages that do not take an argument. They follow the syntactical template receiver methodName. The name of the method is simply a succession of characters not containing : (factorial, open, class). Try: 2000 factorial, Browser open.

Examples 1.13

```
89 sin
3 sqrt
Date today
Double pi
'blop' size
Browser openBrowser
```

> Unary messages are messages that do not take an argument.
> They follow the syntactical template receiver methodName.

Binary Messages. Binary messages are messages that require exactly one argument *and* whose method names are composed of one or two characters from among +, -, *, /, &, =, >, |, <, ~, and @. Note that -- is impossible for parsing reasons.

Examples 1.14

```
100@100
—Printing the returned value:   100@100 "create a point"
3 + 4
—Printing the returned value:   7
total - 1
total <= max
"true if total is less than or equal to max"
(4/3)*3 = 4
—Printing the returned value:   true  "equality is just a binary message, and Fractions are exact"
(3/4) == (3/4)
—Printing the returned value:   false  "two equal Fractions, but not the same object"
```

> Binary messages are messages that require exactly one argument *and* whose method names are composed of one or two characters from +, -, *, /, &, =, >, |, <, ~, and @. -- is not possible. They follow the syntactical template receiver methodName argument.

Keywords-Based Messages. Keyword-based messages are messages that require one or more arguments and have method names including :. Keyword-based messages follow the syntactical template

receiver **methodNameWordOne:** argumentOne **wordTwo:** argumentTwo.

The colon in a method name delimits an argument. Hence r:g:b: is a method with three arguments, playFileNamed:, at: are methods with one argument, and at:put: is a method with two arguments. To create an instance of the class Color, one can use the method r:g:b: as in Color r: 1 g: 0 b: 0, which creates the color red. Note that the colon characters are part of the name of the method.

> In Java or C++, the Smalltalk method invocation Color r: 1 g: 0 b: 0 would be Color.rgb(1,0,0).

Examples 1.15

```
1 to: 10     "create an interval"
Color r: 1 g: 0 b: 0     "create a new color"
MIDIFileReader playFileNamed: 'LetItBe.MID'   "play a midi file"
Speaker manWithEditor say: 'Hello! I hope you will have fun with Squeak'
12 between: 8 and: 15
—Printing the returned value:    true

| array |
array := #(1 2 3).
array at: 1 put: 4.   "example of keyword-based message"
array
—Printing the returned value:    "#(4 2 3)"
```

> Keyword-based messages are messages that require one or more arguments. Their names contains the character :. They follow the syntactical template receiver **methodNameWordOne:** argumentOne **wordTwo:** argumentTwo.

7.1 Message Composition

The three kinds of messages have different orders of precedence, which allows for their composition in an elegant way. Unary messages are always evaluated first, then the binary ones, and finally, the keywords-based ones. Of course, parentheses allow one to change this order. When messages have the same precedence, they are executed from left to right.

> **Rule One.** Unary messages are executed first, then binary messages, and finally keyword-based messages: Unary > Binary > Keyword.

Examples 1.16

```
Bot new go: 30 + 50          "create a robot and move it forward 80 pixels"
Display restoreAfter: [WarpBlt test4]
"Keyword message, try test1, test12, test3, test4, and test 5"
1000 factorial / 999 factorial
—Printing the returned value:    1000
#($t $e $s $t) at: 3
—Printing the returned value:    $s
#($a $b $c $d) at: 2 factorial put: $z
```

As you can see, the syntax rules and in particular the keyword-based messages, as in Example `array at: 1 put: 4`, make it possible to write code with a structure approaching that of natural languages. This was one of the initial objectives in the language design, so that children would be able to program more easily.

> **Rule Two.** Parenthesized messages are executed prior to other messages: `(Msg) > Unary > Binary > Keyword`.

> **Rule Three.** When messages are of the same kind, they are executed from left to right.

Examples 1.17

```
1.5 tan rounded asString "same result as (((1.5 tan) rounded) asString)"
RecordingControlsMorph new openInWorld
"An instance of the digitizer is created then visualized. If your
microphone is plugged in, try a sample!"
3 + 4 factorial
—Printing the returned value:    27 "(not 5040) "
(HTTPSocket httpShowGif:
   'www.altavista.digital.com/av/pix/default/av-adv.gif') display

(FMSound lowMajorScaleOn: FMSound clarinet) play
"display the sound named Clarinet is created then passed as an argument
of the message lowMajorScaleOn:"
```

Mathematical Inconsistency. The message composition rules are simple, but they result in inconsistency with standard mathematical notation in the evaluation of mathematical expressions, because mathematical expressions as just binary messages. Example 1.18 shows the common situations in which extra parentheses are needed.

Examples 1.18

```
3 + 4 * 5
—Printing the returned value:  35 "(not 23)  Binary messages executed from left to right"
3 + (4 * 5)
—Printing the returned value:  23
1 + 1/3
—Printing the returned value:  2/3 "and not 4/3"
1 + (1/3)
—Printing the returned value:  4/3
(1/3) + (2/3)
—Printing the returned value:  1
```

When evaluating the expression 1000 factorial / 999 factorial we obtain 1000. Note that this is an excellent example of automatic coercion and exact handling of a number. Try to display the result of 1000 factorial. It takes more time to display it than to calculate it.

7.2 Keywords Recognition

The final point I want to make here is how keyword-based messages are identified. This will help you understand why you have to use parentheses. The character pairs [,] and (,) delimit distinct areas. In such an area, a keyword-based message is the longest sequence of words terminated by : that is not cut by the characters ., or ;. When the character pair [,] or (,) surrounds some words with colons, these words participate in the keyword-based message *local* to the area defined.

In Example 1.19, there are two distinct keyword-based messages: at:put: and rotatedBy:magnify:smoothing:.

Examples 1.19

```
aDict
  at: (rotatingForm
        rotateBy: angle
        magnify: 2
        smoothing: 1)
  put: 3
```

The characters [,], and (,) delimit distinct areas. In such an area, a keyword-based message is the longest sequence of words terminated by : that is not cut by the characters ., or ;. When the characters [,], and (,) surround some words with colons, these words participate in the keyword-based message local to the area defined.

Hints. You may initially have trouble with these precedence rules. Start simply by using parentheses when you want to distinguish two messages having the same precedence.

Examples 1.20

```
x isNil
  ifTrue:[...]
```

The expression presented in 1.20 does not require parentheses because the expression x isNil is unary and hence is executed prior to the condition.

Examples 1.21

```
| ord |
ord := OrderedCollection new.
(ord includes: $a)
  ifTrue:[...]
```

The expression presented in 1.21 requires parentheses because the messages includes: and ifTrue: are both keyword-based messages. Without parentheses the unrecognized message includes:ifTrue: is sent to the collection.

8 Expression Sequences

Expressions (messages, assignments, ...) separated by periods are executed in sequence. Note that there is no period between a variable definition and the following expression. The value of a sequence is the value of the last expression. The values returned by all the expressions except the last one are ignored. Note that the period is a separator and not a terminating construct. Therefore, a final period is optional.

Examples 1.22

```
| box |
box  := 20@30 corner: 60@90.
box containsPoint: 40@50
```

9 Cascaded Messages

Smalltalk offers a way to send multiple messages to the same object using the construct ;, which is called the cascade in Smalltalk jargon.

```
Expression Msg1 ; Msg2
```

Examples 1.23

```
Transcript show: 'Squeak '.
Transcript show: 'is ' , 'really'.
Transcript show: 'fun '.
Transcript cr.

"is equivalent"

Transcript
  show: 'Squeak ';
  show: 'is ', ' really';
  show: 'fun ';
  cr
```

Note that the object receiving the cascaded messages can be the result of messages. In the following example, the receiver of the cascaded message add: is the newly created ordered collection resulting from the evaluation of OrderedCollection new.

Examples 1.24

```
(OrderedCollection new) add: 25 ; add: 35
```

The previous example is one of the well-known Smalltalk trap. Let's have a look. First you should know that the method add: returns the element added and not the collection; the intermediate results are discarded. Therefore, the result of the complete expression is the one of the last cascaded message add: 35, that is 35. We need a way to return the receiver of the cascaded messages, here the created ordered collection. In fact, we need a way to return self. This is the sole purpose of the method yourself. Example 1.25 returns the ordered collection containing the elements 25 and 35.

Examples 1.25

```
(OrderedCollection new) add: 25 ; add: 35 ; yourself
```

10 Primitives

In Smalltalk everything is an object, and computation is done by executing methods in response to messages. However, not everything can be done within this context. The system should rely on a small number of methods that are defined in the underlying implementation language (C, C++, or assembly) of the virtual machine. Certain methods invoke these functionalities that are called *primitives*. As a Smalltalk programmer you do not have to worry about whether the method you invoke is a primitive.

To give you an idea of primitives, the following functionality is implemented as primitive calls: memory allocation (new, new:), bit manipulation (bitAnd:, bitOr:, bitShift:), pointer and integer arithmetic (+, -, <, >, *, / , =, ==, ...), and array access (at:, at:put:). Such methods call primitives using the <primitive: aNumber> construct. After this construct there is the code that is evaluated when the primitive fails. In the case of primitives that cannot be expressed in the object paradigm, the method primitiveFailed is called.

Method 1.2

```
ProtoObject>>== anObject
  "Primitive. Answer whether the receiver and the argument are the same
  object (have the same object pointer). Do not redefine the message == in
  any other class! Essential. No Lookup. Do not override in any subclass.
  See Object documentation whatIsAPrimitive."

  <primitive: 110>
  self primitiveFailed
```

The other use of primitives is to optimize some crucial methods. The idea is that the system could work without the primitive but it would be slow. The following method shows that the method @ calls the primitive 18. Here the point creation is clearly expressible in Smalltalk. Therefore, the code after the primitive is just the creation of a point illustrating what the primitive is actually doing. Note that such code will be never called except if the primitive would have failed, which is extremely rare.

Examples 1.26

```
Integer>>@ y
  "Primitive. Answer a Point whose x value is the receiver and whose y
  value is the argument. Optional. No Lookup. See Object documentation
  whatIsAPrimitive."

  <primitive: 18>
  ^Point x: self y: y
```

11 Block-Closures

Block-closures constitute an extremely powerful feature of Smalltalk inherited from functional programming languages. Block-closures are anonymous methods or sequences of messages that are deferred. In its pure functional form a block is similar to a mathematical function. Blocks are instances of the class BlockContext. The function $x \rightarrow 2x + 1$ is represented by the block [:x| 2*x + 1]. $f(x) \rightarrow 2x + 1$ can be represented as |f| f:=[:x| 2*x + 1]. $f(3)$ is f value: 3.

Examples 1.27

```
|f|
f := [ :x| 2*x + 1 ].
f value: 3
—Printing the returned value:  7
```

Note for the Java Programmer. Innerclasses were introduced in java because the language did not offer block-closures.

A block can have zero or multiple arguments defined between the [and | characters. They start with a : character. Block can also define local variables delimited by |, and then have a sequence of messages. The value of a block is the value of its last expression except if there is an explicit return ^ .

Important Messages 1.1

```
[ :variable1 :variable2 |
   | blockTemporary1 blockTemporary2 |
   expression1.
   ...variable1 ...
   expressionz ]
```

Examples 1.28

```
[ 2 + 3 + 4 + 5 ]                  "a Block with  no argument"
[ :x | x + 3 + 4 + 5 ]              "a Block with  1 argument"
[ :x :y | x + y + 4 + 5 ]           "a Block with  2 arguments"
[ :x :y :z | x + y + z + 5 ]        "a Block with  3 arguments"
[ :x :y :z :w | x + y + z + w ]     "a Block with  4 arguments"
```

A block is a deferred sequence of messages, in the sense that the messages it contains are not executed when the block is created. To get the block executed and obtain its value, we use the methods value, value:, value:value:, ... If you have a block with more than four parameters, you must use valueWithArguments: and pass the argument in an array. Note that this often reveals a design problem.

Examples 1.29

```
[ 2 + 3 + 4 + 5 ] value
[ :x | x + 3 + 4 + 5 ] value: 2
[ :x :y | x + y + 4 + 5 ] value: 2 value: 3
[ :x :y :z | x + y + z + 5 ] value: 2 value: 3 value: 4
[ :x :y :z :w | x + y + z + w ] value: 2 value: 3 value: 4 value: 5
```

In fact, block-closures are functions that capture of the environment in which there are defined, that is they can refer to variables that are not defined in the context of the block. That is why they are called block-*closures*. Because they perform a lexical closure, they can refer to the variables that were visible in the context where the block was defined. Note however that Squeak contrary to other smalltalks does not have full closure as the block variables are stored in the method defining the block, leading sometimes to problems.

Examples 1.30

```
| index bloc |
index := 0.
bloc := [ index := index +1 ].
bloc value
—Printing the returned value:    1
index := 3.
bloc value
—Printing the returned value:    4
```

Blocks can be assigned to variables and passed as method arguments. Note that for performance reasons. it is always better to avoid reference to variables external to the block or to have explicit returns. Writing blocks that only refer to their arguments lets the compiler optimize them considerably.

Squeak in its version 3.4 does not really support real block-closures. Block-closures are simulated (block arguments are defined as temporary methods of the methods that define the block) and not implemented as in all other versions of Smalltalk. This is due to the fact that Squeak is based on one of the first implementations of Smalltalk. This situation will probably change because new implementations supporting real block-closures are under development at the time of this writing.

12 Return Expressions

Returning a value is done by prefixing an expression with the character ^ . Inside the body of a method or a block, a return expression is used to terminate the execution of the method and deliver the expression as the method's result.

Really Important. The character ^ acts as an escape mechanism. Return expressions inside a nested block expression will terminate the enclosing method, as shown in Example 1.31.

Examples 1.31

```
when the expression ^ x@y is executed, the method detect:
escapes the current iteration and returns it.

TwoLevelSet>>detect: aBlock

firstLevel keysAndValuesDo: [ :x :v |
v do: [ :y | (aBlock value: x@y) ifTrue: [ ^x@y ] ] ].
^nil
```

13 Control Structures

As I already have mentioned, in Smalltalk there are no dedicated constructs with a particular syntax for loops, conditionasl, or exceptions. Keyword-based messages with block-closures suffice to express them.

13.1 Conditional Control Structures (Receiver Is a Boolean)

There are four methods defined on the classes Boolean, True, and False that define conditional execution: ifTrue:ifFalse:, ifFalse:ifTrue:, ifTrue:, and ifFalse:.

These methods have the following behavior:

- aBooleanExpression ifTrue: trueAlternativeBlock ifFalse: falseAlternativeBlock and ifFalse: falseAlternativeBlock ifTrue: trueAlternativeBlock return the value of trueAlternativeBlock if the receiver is true. Return the value of falseAlternativeBlock if the receiver is false. Otherwise, signal an error.
- aBooleanExpression ifTrue: alternativeBlock returns nil if the receiver is false. Signal an error if the receiver is not a Boolean. Otherwise, return the result of evaluating alternativeBlock.
- Symetrically aBooleanExpression ifFalse: alternativeBlock returns nil if the receiver is true. Signal an error if the receiver is not a Boolean. Otherwise, return the result of evaluating the argument alternativeBlock.

Note that the blocks used for conditionals are blocks without arguments, and like any block they can contain sequences of messages. All these conditional expressions are just methods defined on Boolean classes. Note that the compiler is optimizing them drastically, so that the messages are not even sent.

Examples 1.32

```
10 factorial isOdd
  ifTrue: [ Transcript show: 'odd Number'; cr ]
  ifFalse: [ Transcript show: 'even Number'; cr.
            Beeper beep ]
```

Composing Boolean Expressions. Boolean classes also define methods for composing Boolean expressions: & aBoolean (and), and: aBlock (lazy and), | aBoolean (or), or: aBlock (lazy or), not (negation), and xor: aBlock (exclusive or) (see Chapter **??**.

13.2 Iterative Conditional Structures (Receiver Is aBlockContext)

Besides conditional expressions, the class BlockContext defines the following methods that implement iterative conditional control structures: whileTrue, whileTrue:, whileFalse, and whileFalse:.

These methods have the following behavior:

- aBlock whileTrue evaluates the receiver and continues to evaluate the receiver as long as the result is true.
- aBlock whileTrue: anotherBlock evaluates the receiver. If the receiver returns true, evaluates anotherBlock and repeats as shown in Example 1.33.
- whileFalse and whileFalse:

Examples 1.33

```
| counter |
counter := 0.
[ counter < 100 ] whileTrue:
   [ counter := counter + 1.
   Beeper beep ]
```

The definition of the methods whileTrue: and whileTrue is another example of the use of conditional loops (see method 1.3). Note again that for the methods whileTrue and whileTrue:, the compiler heavily optimizes their invocation to the point that they are not executed in this current form.

Method 1.3

```
BlockClosure>>whileTrue: aBlock
  ^ self value
      ifTrue:
        [ aBlock value.
        self whileTrue: aBlock ]

BlockClosure>>whileTrue
  ^ [ self value ] whileTrue: []
```

13.3 Enumeration Control Structures (Receiver Is anInteger)

The class Integer defines the following unconditional loops: timesRepeat:, to:do:, and to:by:do:. They have the following behavior:

- startInteger timesRepeat: aBlock evaluates the argument, aBlock, the number of times represented by the receiver.
- startInteger to: stopInteger do: aOneArgBlock evaluates aOneArgBlock with each element of the interval (self to: stop by: 1) as argument, as illustrated in Example 1.35, which defines factorial.
- startInteger to: stopInteger by: stepInteger do: aOneArgBlock evaluates aOneArgBlock with each element of the interval (self to: stop by: step) as argument.

Examples 1.34

```
4 timesRepeat: [ Beeper beep ]
1 to: 3 do: [ :i | Transcript show: i printString ;cr ]
"This sends #to:do: (with two parameters) to integer 1"
(1 to: 3) do:  [ :i | Transcript show: i printString ;cr ]
"This sends #do: to the Interval given by evaluating '1 to: 3'"
```

Examples 1.35

```
| tmp |
tmp := 1.
2 to: self do: [ :i | tmp := tmp * i ].
tmp
```

13.4 Enumeration Control Structures (Receiver Is Collection)

Smalltalk also defines extremely powerful methods for iterating over collections that have copied in other languages, namely do:, collect:, select:, reject:, detect:, detect:ifNone:, with:do:, inject:into:.

The behavior of these methods is the following:

- aCollection do: aOneArgBlock applies aOneArgBlock, a block requiring one argument, to each collection element.
- aCollection collect: aOneArgBlock returns a collection containing all the results obtained by the application of the block aOneArgBlock to each collection element.
- aCollection with: anotherCollection do: aBinaryBlock iterates in parallel over each element of the two collections and executes the block requiring two arguments with the elements of each collection.

Examples 1.36

```
#(15 10 19 68) do:
  [ :i | Transcript show: i printString ; cr ]

#(15 10 19 68) collect: [ :i | i odd ]
  —Printing the returned value:    #(true false true false)

#(1 2 3) with: #(10 20 30)
do: [ :x :y | Transcript show: (y ** x) printString ; cr ]
```

- aCollection select: aOneArgPredicateBlock returns a collection representing all the elements of the collection satisfying the block aOneArgPredicateBlock. A block should take one argument and return a Boolean.

- aCollection reject: aOneArgPredicateBlock returns a collection representing all the elements of the collection not satisfying the block aPredicateBlock.

- aCollection detect: aOneArgPredicateBlock returns the first element satisfying the block aOneArgPredicateBlock. Raise an error if none of the elements satisfy the block.

- aCollection detect: aOneArgPredicateBlock ifNone: aNoneBlock returns the first element satisfying the block aOneArgPredicateBlock. If no element satisfies it, returns the value returned by the aNoneBlock.

Examples 1.37

```
#(15 10 19 68) select: [ :i | i odd ]
—Printing the returned value:    #(15 19)
#(15 10 19 68) reject: [ :i | i odd ]
—Printing the returned value:    #(10 68)
#(12 10 19 68 21) detect: [ :i | i odd ]
—Printing the returned value:     19
#(12 10 12 68) detect: [ :i | i odd ] ifNone:[ 1 ]
—Printing the returned value:    1
```

- aCollection inject: aStartValue into: aBinaryBlock applies the block aBinaryBlock to all the elements of the receiver, accumulating the result in the first argument starting with aStartValue.

Examples 1.38

```
|acc|
acc := 0.
#(1 2 3 4 5) do: [ :element | acc := acc + element ].
acc
—Printing the returned value:    15

"Is equivalent to"

#(1 2 3 4 5)
  inject: 0
  into: [ :acc :element | acc + element ]
```

14 Exceptions

Squeak supports ANSI standard exceptions. Exceptions are objects too. Exceptions are again not another construct. They are just methods associated with blocks. To catch an exception use the method on: anException do: aUnaryBlock. To raise an exception use the method signal defined on the class Exception.

Examples 1.39

```
[| tmp |
10 to: -10 by: -1 do: [ :i | Transcript show: (100 / i) printString ; cr ] ]
  on: ZeroDivide
  do: [:ex |  ]
```

The most important messages defined on the class BlockContext are ensure: aTermination-Block, which evaluates aTerminationBlock after evaluating the receiver, regardless of whether the receiver's evaluation completes; ifCurtailed: aTerminationBlock, which evaluates the receiver, and if it terminates abnormally, evaluates aTerminationBlock; and on: exception do: handlerAction-Block, which evaluates the receiver in the scope of an exception handler, handlerActionBlock.

Examples 1.40

```
["target code, which may abort"]
  ensure:
    ["code that will always be executed after the target code,
    whatever may happen"]

["target code, which may abort"]
  ifCurtailed:
    ["code that will be executed whenever the
    target code terminates without a normal return"]

["target code, which may abort"]
  on: Exception
  do:
    [:exception |
    "code that will be executed whenever the identified Exception
    is signaled."]
```

Exceptions have rich possibilities, such as being resumed or restarted, or signaling other exceptions. Browse the class Exception if you want to know more.

15 Class Definition

There is no specific construct to define classes in Smalltalk. Classes are created by sending messages to the superclass of the class we want to create. Several kinds of classes can be created depending on whether their instances have fixed size or their instances can have arbitrary size such as arrays do.

Ordinary Class Definition. The first kind of class is created using the message subclass:instanceVariableNames:classVariableNames:poolDictionaries:category:, and this is the way you will create classes in 99% of the cases. Instances of such classes have named instance variables, hence a precise size.

Class 1.1

```
SuperClass subclass: #NameOfClass
   instanceVariableNames: 'instVarName1 instVarName2'
   classVariableNames: 'ClassVarName1 ClassVarName2'
   poolDictionaries: ''
   category: 'Major-Minor'
```

Variable Class Definition. The second kind of class is created using one of the following three messages: variableSubclass:..., variableByteSubclass:..., variableWordSubclass:.... These kinds of classes are used to create indexable objects, such as instances of Array, ByteArray, and WordArray. The instances of such a class have a size that is arbitrary and is specified at the instance creation using the message new: anInteger. The different messages indicate the kind of indexable entity: variableSubclass:... specifies that instances have indexed variables that contain object pointers, while

variableByteSubclass:... specifies that instances have indexed variables that contain bytes, and that instances have indexed variables that contain words. I have included them here for completeness, but it is extremely rare that you will need them.

Class 1.2

```
SuperClass variableSubclass: #NameOfClass
   instanceVariableNames: 'instVarName1 instVarName2'
   classVariableNames: 'ClassVarName1 ClassVarName2'
   poolDictionaries: ''
   category: 'Major-Minor'

SuperClass variableByteSubclass: #NameOfClass
   instanceVariableNames: 'instVarName1 instVarName2'
   classVariableNames: 'ClassVarName1 ClassVarName2'
   poolDictionaries: ''
   category: 'Major-Minor'

SuperClass variableWordSubclass: #NameOfClass
   instanceVariableNames: 'instVarName1 instVarName2'
   classVariableNames: 'ClassVarName1 ClassVarName2'
   poolDictionaries: ''
   category: 'Major-Minor'
```

16 Method Definition

Methods are defined by writing the method signature followed by the method body in the browsers and compiling them using the **accept menu** item. The class in which the method is defined is not explicitly mentioned, as shown by Method 1.4. The browsers compile a method in the class in which the method is defined. Note that there exist a couple of messages to explicitly ask the class to compile a method, but these messages are used by the compiler or the environment such as the browser or the file list to create methods. A normal programmer does not need to know them.

Methods always return a value; there is an implicit ^ self at the end of every method to make sure that this is the case. In Method method 1.4 (from String), the method returns a value explicitly, while in Method method 1.5, defined on the class Browser, there is no explicit return value, so the method returns self, which here represents the class itself, since the method is a class method.

Method 1.4

```
lineCount
  "Answer the number of lines represented by the receiver, where every
  cr adds one line."

  | cr count |
  cr  Character cr.
  count  1  min: self size.
  self do:
    [ :c | c == cr ifTrue: [count  count + 1] ].
  ^ count
```

Method 1.5

```
openBrowser
  "Create and schedule a BrowserView with default browser label. The
  view consists of five subviews, starting with the list view of system
  categories of SystemOrganization. The initial text view part is empty."

  | br |
  br := self new.
  self
    openBrowserView: (br openEditString: nil)
    label: br defaultBrowserTitle
```

17 What You Should Know

The syntax of Squeak is based on the following:

1. six reserved identifiers, also called *pseudovariables*: true, false, nil, self, super, and thisContext;
2. constant expressions called *literal objects*: Numbers, Characters ($a), Strings ('aString yes!'), Symbols that are unique strings #aUniqueSymbol, Arrays (#(a #(b 1)));
3. identifiers and assignment (:=, _, or ←);
4. dynamically created arrays, also called brace arrays ({1+2 . 3*5});
5. *messages*, which are of three kinds: *unary* (asString, new), *binary* (+, <=, &, @, ...), and *keyword-based* messages (at:, at:put:, r:g:b:). Unary messages have the highest precedence, then binary, then keyword-based. Messages with the same precedence are evaluated from left to right;
6. *block-closures* are anonymous and deferred sequences of messages ([:x | x + 2] value: 4).

Based on these syntactical elements, conditional expressions (e.g., ifTrue:ifFalse:, whileTrue:), iterations (e.g., timesRepeat:, to:do:, do:, collect:, select:), and exceptions (e.g., on:to:, ifCurtailed:) are naturally expressed without requiring any dedicated constructs.