CS 683 Emerging Technologies: Embracing Change
Spring Semester, 2001
Doc 2 Starting Squeak
Contents

Simple Basic Types....2
Boolean....2
nil....2
Object....3
Equality....5
Numbers....7
Integers....7
Float....8
Fraction....9
Some Numerical Methods....10
Characters....11
Blocks....13
Creating Processes (or Threads)....18
Control Messages....20
If....20
Boolean Expressions....21
Basic Loops....27

References

Squeak Smalltalk: A Quick Reference, Andrew C. Greenberg, http://www.mucow.com/squeak-gref.html

Object-Oriented Design with Smalltalk — a Pure Object Language and its Environment, Ducasse, University of Bern, Lecture notes 2000/2001, http://www.iam.unibe.ch/~ducasse/WebPages/Smalltalk/ST00_01.pdf

Smalltalk Best Practice Patterns, Kent Beck, Prentice Hall, 1997

Reading

Squeak: Object-Oriented Design with Multimedia Applications, Guzdial, Chapter 1, and Chapter 2 sections 2.1- 2.3

Copyright ©, All rights reserved. 2001 SDSU & Roger Whitney, 5500 Campanile Drive, San Diego, CA 92182-7700 USA. OpenContent (http://www.opencontent.org/opl.shtml) license defines the copyright on this document.

Simple Basic Types
Boolean

true
Unique instance of True class

false
Unique instance of the False class

Smalltalk uses true and false for boolean values

Boolean operators (2 > 10) result in true or false

Integers (0, 1, etc) can not be used for boolean values

nil

Value of an uninitialized variable

Unique instance of the UndefinedObject class

Object

All 'things' in Smalltalk are objects

Objects are created from classes

The class Object is the parent class of all classes

Object class contains common methods for all objects

Determines behavior for all objects

Some Important Behavior Defined in Object

printString

Returns a string representation of the receiver

Similar to toString in Java

isNil, notNil

Tests to see if the receiver has been initialized or is still nil

class

Returns the class of the receiver

```
| a |                    Result
a isNil.                 true
a printString.           'nil'
a class.                 UndefinedObject
a := 5.
a isNil.                 false
a printString.           '5'
a class                  SmallInteger
```

When the above code is compiled the compiler notices that a is used before it is assigned a value. The compiler will ask you if want use a before it is assigned.

Equality

All objects are allocated on the heap

Variables are references (like a pointer) to objects

A == B
Returns true if the two variables point to the same location

A = B
Returns true if the two variables point to objects that logically represent the same object.

In Smalltalk you want to use '=' nearly all the time

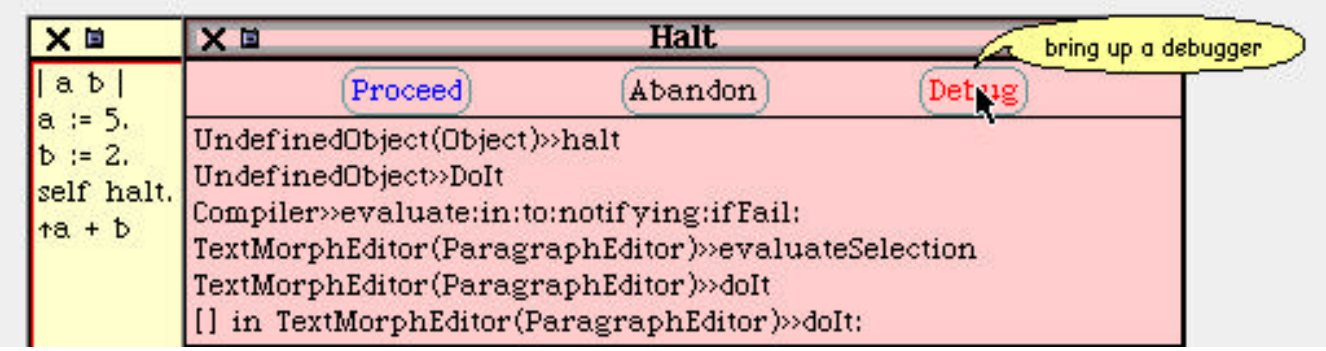
A ~= B
Means (A = B) not

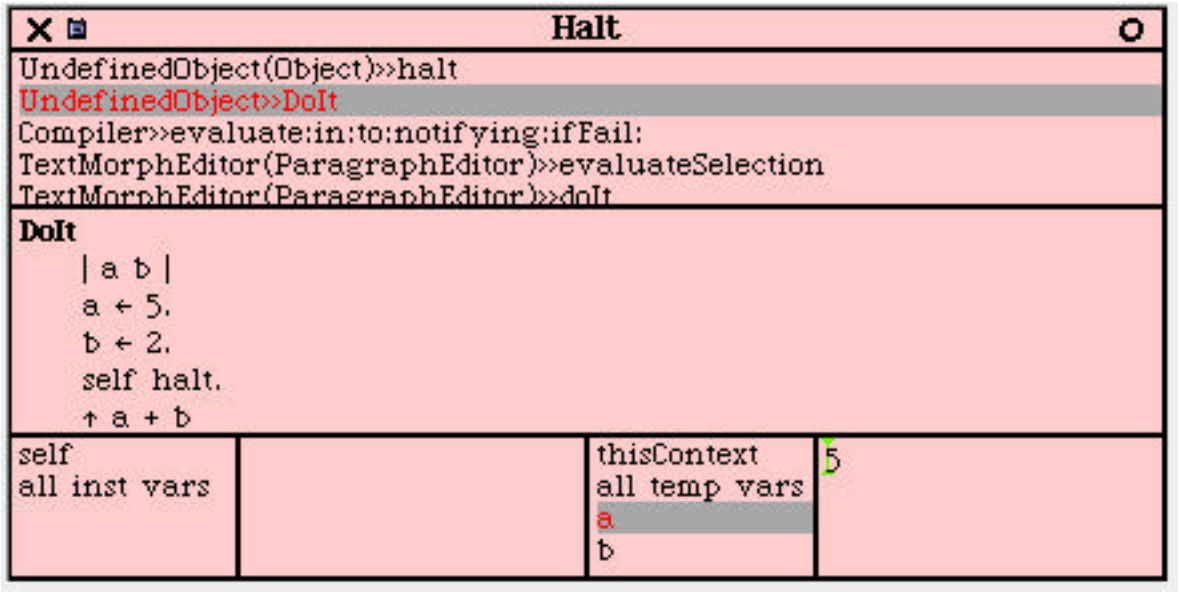
A ~~ B
Means (A == B) not

halt

Open the debugger

```
| a b |
a := 5.
b := 2.
self halt.          "self is like this is Java/C++, more later"
^a + b
```





The debugger is very useful. We will see more of this later.

Numbers
Integers

Smalltalk supports Integers of arbitrary size

Available memory dictates integer range

-1073741824 to 1073741823 (30 bits) are handled efficiently

Integers larger than 30 bits require multiple words

Literal Forms

```
1234
12345678901234567890123456789012345678901234567890123456
```

With exponent

Expression	Value
123e2	12300

With base <base>r<number>

Expression	Value
16rFF	255
8r11	9
3r120	15

With base and exponent

16rFFe2	65280
8r11e3	4608

Float

IEEE-754 floating-point double precision numbers
About 16 digits of accuracy
Range ±10^307

Literal forms

Expression	Value
12.34	12.34
123e2	12300
3.14e-10	3.14e-10
16rFF.1Ae2	65306.0
8r11.4e3	4864.0

Infinity & NaN

IEEE-754 supports infinity and NaN (Not a Number)
Some float operations may result in these values

Expression	Result
12.5e300/0.1e-50	Infinity
-12.5e300/0.1e-50	-Infinity
(12.5e300/0.1e-50) isInfinite	true
(12.5e300/0.1e-50) isNaN	false
(12.5e300/0.1e-50) isFloat	true

Some Constants

Float e	2.718281828459045
Float halfPi	1.570796326794897
Float negativeZero	-0.0
Float pi	3.141592653589793

Fraction

Integer division results in a fraction

Expression	Result
1/2	(1/2)
(1/2) + (1/3)	(5/6)
(2r11/16rAA) * 2	(3/85)

Converting Fractions to Floats

Operations with floats convert fractions to floats

The asFloat message converts a fraction to a float

Expression	Result
1/2 asFloat	0.5
(1/2) + 1.5	2.0
(1.0/3)	0.3333333333333333

Some Numerical Methods

Arithmetic

* + - / // \\ abs negated quo: reciprocal rem:

		Result
division	4/2	4/2
integer division	5//2	2
modulo	5\\2	1
	-3 abs	3
	5 negated	-5

Rounding

4.2 ceiling	5
4.2 floor	4
3.1523 roundTo: 0.01	3.15
4.2 truncated	4

Testing

3.2 even	false
22.2 isDivisibleBy: 1.1	false
-3 sign	-1
5.1 closeTo: 5.1000001	true

odd, isInfinite, isNaN, isNumber, isZero,
negative, odd, positive, strictlyPositive,

Others

arcCos, arcSin, arcTan, arcTan:, cos, exp,
floorLog:, ln, log, log:, raisedTo:, raisedToInteger:,
sin, sqrt, squared, tan

Characters

Various ways to reference a single character

```
| aChar |
aChar := $a.
aChar := $5.
aChar := Character tab.
aChar := Character value: 65.
aChar := 65 asCharacter.
aChar := 'cat' at: 1.              "indexing starts at 1"
```

Character class provides names for white space characters

backspace	cr	enter
escape	euro	lf
linefeed	nbsp	newPage
tab		

Some Character Operations

asciiValue	digitValue	<	=
>	isAlphaNumeric	isDigit	isLetter
isLowercase	isSafeForHTTP	isSeparator	isSpecial
isUppercase	isVowel	asCharacter	asIRCLowercase
asInteger	asLowercase	asString	asSymbol
asUppercase			

$a isVowel return true

What about Strings?

Squeak does have strings. Some important string methods use blocks. So we will first cover blocks. We will get back to strings.

Blocks

- A deferred sequence of actions
- Can have 0 or more arguments
- Executed when sent the message 'value'

Similar to
- Lisp Lambda- Expression
- C function
- Anonymous functions

General Format

```
[:variable1 :variable2 … :variableN |
    | blockTemporary1 blockTemporary2 … blockTemporaryK |
    expression1.
    expression2.
    …]
```

Zero Argument Block

```
| block x |
x := 5.
block := [Transcript show: x printString].
x := 10.
block value
```

Prints 10 in the Transcript window

```
| block x |
x := 5.
block := [:argument | Transcript show: (x + argument) printString].
x := 10.
block value: 4
```

Prints 14 in the Transcript window

Blocks and Return Values

Blocks return the value of the last executed statement in the block

```
| block x |
block := [:a :b |
                | c |
                c := a + b.
                c + 5].

x := block value: 1 value: 2.
```

x has the value 8

Blocks and Arguments

```
[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
```

Using the value: keyword message up to 4 arguments can be sent to a block.

```
[:a :b :c :d :e | a + b + c + d + e ] valueWithArguments: #( 1 2 3 4 5)
[:a :b | a + b ] valueWithArguments: #( 1 2 )
```

With the keyword message valueWithArguments: 1 or more arguments can be sent to a block

The argument to valueWithArguments: must be an array

#(1 2 3) creates an array.

More on arrays soon.

But what are Blocks Good for?

The examples of blocks so far are not very useful (except to show the syntax of blocks). Blocks are one of Smalltalk's strong points. We will look at some uses of blocks: threads, timing code, and what most languages call control structures. After that we will cover Arrays and Collections. There we will cover iteration, which also uses blocks.

Creating Processes (or Threads)

[code] fork
fork a new process to execute the block

Process runs at same priority as current process

[code] forkAt: anInteger
fork a new process at priority anInteger to execute the block

Priorities range from 1 (low) to 8 (high)

Example

```
[Transcript show: 'hi'] fork.
Transcript show: 'bye'
```

Output in Transcript

```
byehi
```

Processes in Squeak are lightweight. That is processes run in the same memory space. Java uses the term thread for lightweight processes. Until we cover more of the language we can't do much with processes. It is nice to know that they exist are easy to start. However, debugging multithreaded programs in any language can be a challenge.

Timing Code

[code] timeToRun

Returns the milliseconds it takes to execute the block

Example

[3 + 4] timeToRun

Result

0

Until we know how to perform loops etc we can not write code interesting enough to time. So on to what most languages call control structures.

Control Messages
If

Format (4 versions)

(boolean expression) ifTrue: trueBlock

(boolean expression) ifFalse: falseBlock

(boolean expression) ifFalse: falseBlock ifTrue: trueBlock

(boolean expression) ifTrue: trueBlock ifFalse: falseBlock

Examples

```
difference := (x > y)
        ifTrue: [ x - y]
        ifFalse: [ y - x]


a < 1 ifTrue: [Transcript show: 'hi mom' ]

x sin < 0.5 ifTrue:
    [y := x cos.
    z := y + 12.
    w := z cos]
```

Boolean Expressions
Logical Operations

	Symbol	Example
Or	|	a | b
And	&	a & b
Exclusive OR	xor:	a xor: (b > c)
Negation	not	(a < b) not

Lazy Logical Operations

	Message	Example
Or	or: orBlock	a or: [b > c]
And	and: andBlock	a and: [c | b]

The orBlock is evaluated only if the receiver of or: is false

The andBlock is evaluated only if the receiver of and: is true

Where is the Value Message?

In the message:

```
difference := (x > y)
        ifTrue: [ x - y]
        ifFalse: [ y - x]
```

where is value sent to the blocks?

In the False class we have:

```
ifTrue: trueAlternativeBlock ifFalse: falseAlternativeBlock
    ↑ falseAlternativeBlock value
```

In the True class we have:

```
ifTrue: trueAlternativeBlock ifFalse: falseAlternativeBlock
    ↑ trueAlternativeBlock value
```

The value message is send to the correct block in the True or False class depending on the value of (x > y)

Performance Note

To improve performance the compiler inlines some messages.

Since it does not make sense to send ifTrue: to anything but true and false, ifTrue: and ifFalse: messages are inlined. So they look like messages and they seem to act like messages, they do have the overhead of messages. One does not realize this unless one tries to modify the ifTrue: ifFalse: methods in the True and False classes. The changes would not have any effect. Since Squeak is implemented in Squeak if one could really change the ifTrue: & ifFalse: methods, the system would either crash or behave very bizarrely.

Can I send ifTrue: to a non-Boolean?

Smalltalk compilers do not check for type usage

Type usage is check at runtime

If you send a message to an object that it does not implement a runtime error results

So if you execute the following you get a runtime error not a compile error:

```
5 ifTrue: [1 + 3 ]
```

Type Checking Verses Runtime Checking

A number of people believe that large programs can not be written in languages without typing, preferable strong type checking. They believe that without the compiler checking type usage programmers will make too many type usage errors. This will slow the development of programs and result in too many errors. Programmers using Smalltalk, Lisp, Perl, APL, Python or Ruby (to name a few) tend to believe that type usage slows program development. Mixing these two groups of people in newsgroups results in many flame wars. These flame wars are a waste of emotional energy. Try Smalltalk and see for yourself. You might find that for you type checking is very important. If so then you know it by experience rather than repeating what you were told in a course. You might find that you do just fine without type checking.

A Style Issue

Both of the following have the same effect

Which to use?

```
difference := (x > y)
        ifTrue: [ x - y]
        ifFalse: [ y - x]
```

```
(x > y)
    ifTrue: [difference := x - y]
    ifFalse: [difference := y - x]
```

Smalltalkers use and prefer the first version[1].

The main goal of the above statements is to assign a value to difference. The first statement makes this clear. The second statement makes you work to see the both paths of the computation assign a value to difference

[1] See Smalltalk Best Practice Patterns, Kent Beck, Conditional Expression Pattern, page 180.

IfNil:

Format

ifNil: aBlock

ifNotNil: aBlock

ifNil: nilBlock ifNotNil: notNilBlock

ifNotNil: notNilBlock ifNil: nilBlock

The following expression is very common in Smalltalk

```
x isNil
    ifTrue: [ some code ]
    ifFalse: [ other code ]
```

Squeak has a short hand version ifNil

```
x
    ifNil: [ some code ]
    ifNotNil: [ other code ]
```

Basic Loops

Format

```
aBlockTest whileTrue
aBlockTest whileTrue: aBlockBody
aBlockTest whileFalse
aBlockTest whileFalse: aBlockBody
```

The last expression in aBlockTest must evaluate to a boolean

Examples

```
| x y difference |
x := 8.
y := 6.
difference := 0.
[x > y] whileTrue:
      [difference := difference + 1.
      y := y + 1].
↑difference
```

```
| count |
count := 0.
[count := count + 1.
 count < 100] whileTrue.
Transcript clear; show: count printString
```

Note that with the whileTrue: message we can perform the loop check before we enter the loop, like a while statement in C/C++/Java. The whileTrue message acts like the do while statement in Java.

More Loops

Format

anInteger timesRepeat: aBodyBlock

startInteger to: endInteger do: blockWithArgument

start to: end by: increment do: blockWithArgument

```
Transcript
    open;
    clear.
3 timesRepeat:
    [Transcript
        cr;
        show: 'Testing!'].

1 to: 3 do:
    [ :n |
    Transcript
        cr;
        show: n printString;
        tab;
        show: n squared printString].

9 to: 1 by: -2 do:
    [ :n |
    Transcript
        cr;
        show: n printString].
```