CS 535 Object-Oriented Programming & Design
Fall Semester, 2001
Doc 2 Basic Smalltalk Syntax
Contents

References

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

Basic Smalltalk Syntax

The Xerox team spent 10 years developing Smalltalk

They thought carefully about the syntax of the language

Smalltalk syntax is

- Different from other languages
- Simple and compact

The Rules

- Everything in Smalltalk is an object
- All actions are done by sending a message to an object
- Every object is an instance of a class
- All classes have a parent class
- Object is the root class

Entire Language Example
From Ralph Johnson

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

    | 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
```

Sample Program

```
"A Sample comment"

| a b |
a :='this is a string'.                    ":= is assignment"
a :=  'this is " a string that contains
      a single quote and a newline'.

a := 'concat' , 'inate'.
a := 5.
a := 1 + "comments ignored"  1.

b := 2 raisedTo: 5.
_a + b                                  "_ (up arrow) means return"
```

Multiple Assignments

Assignment statements return values!

```
| a b |
a := b := 3 + 4.
```

a and b now contain 7

Statement Separator

```
| cat dog |
cat := 5.
dog := cat + 2
```

A period is used as a statement separator

A period is optional after the last statement

Identifiers

An identifier (any name) in Smalltalk is of the form:

letter (letter | digit)*

Variables

```
| cat dog |
cat := 5.
dog := cat + 2.
```

Vertical bars at the top of a program declare variables

Variables must be declared

All variables are references to objects

Variables are initialized to nil

As we will see numbers in Smalltalk are objects. Internally references to objects require pointers. Always using a pointer to refer to a number would slow arithemitic operations. Most Smalltalk virtual machines will store numbers directly in a variable. At the programming level one does not see any difference in how numbers and other objects are handled.

Messages

Most languages place basic operations in the grammar

>, =, for (int k = 1; k < 10; k++)

In Smalltalk operations are defined as methods in a class

+ is a method in the Integer class

In 3 + 4, + is a message sent to the integer 3

Using messages rather than hard coded grammar makes

- Parsing code simpler
- Language extensible

Three type of Messages

- Binary

  1 + 2
  12 / 6

- Unary

  12.3 printString
  '123' asNumber

- Keyword

  'Hi mom' copyFrom: 1 to: 3

All messages contain:

- Receiver
- Selector
- Zero or more arguments

Messages always return a value

Unary Messages

Format:
aReceiver aSelector

25 factorial

25 is the receiver
factorial is the selector
returns 15511210043330985984000000

'this is a string' reversed

'this is a string' is the receiver
reversed is the selector
returns 'gnirts a si siht'

'Cat in the hat' size

returns 14

12 printString

returns ‘12’ (a string)

‘20’ asNumber

returns 20 (an integer)

Combining Unary Messages

Unary messages are executed from left to right

```
100 factorial printString size
```

is done as:

```
((100 factorial) printString) size
```

How about this?

```
100 factorial size
```

This will not work

100 factorial returns an integer

Integers do not implement a size method

One uses the Smalltalk browser to see what methods a class implements

Binary Messages

Format:

aReceiver aSelector anArgument

2 + 4

2 is the receiver
+ is the selector
4 is the argument
returns 6

Binary selectors are

- Arithmetic, comparison and logical operations
- One or two characters taken from:

  + - / \ * ~ < > = @ % | & ! ? ,

  Second character is never -

Using the above rules you can create your own binary messages in Smalltalk. You can make @? a binary method in a class.

Combining Binary Messages

Binary messages are executed from left to right

1 + 2 * 3 * 4 + 5 * 6

is executed as

((((1 + 2) * 3) * 4) + 5) * 6

Keyword Messages

Format:
receiver keyword1: argument1 keyword2: argument2 …

12 min: 6

12 is the receiver
min: is a selector with only one keyword
6 is the argument
returns 6

```
'this is a string'
    copyFrom: 1
    to: 7
```

'this is a string' is the receiver
copyFrom:to: is one selector with two keywords
1 and 7 are the arguments
returns 'this is'

```
'this is a string'
    findString: 'string'
    startingAt: 4
    ignoreCase: true
    useWildcards: false
```

'this is a string' is the receiver
findString:startingAt:ignoreCase:useWildcards: is one selector
‘string’, 4, true, false are the arguments
returns (11 to: 16)

Keyword Messages verses Positional Argument Lists Smalltalk Version

```
'this is a string'
        findString: 'string'
        startingAt: 4
        ignoreCase: true
        useWildcards: false
```

- Each keyword communicates role of argument

Positional Argument List Version

```
'this is a string'.findString( 'string', 4, true, false);
```

- More common so more familiar
- Easy for compiler to parse
- Easier for programmer to mix up parameters

Where do Keyword Messages End?

Unless you use parenthesis the compiler combines all keywords in a statement into one message

```
'this is a string'
      copyFrom: 1
      to: 12 min: 7
```

The above has one message

copyFrom:to:min:

This message does not exist, so results in an error

```
'this is a string'
      copyFrom: 1
      to: (12 min: 7)
```

This message contains two legal keyword messages

Formatting Keyword Messages

```
'this is a string'
    findString: 'string'
    startingAt: 4
    ignoreCase: true
    useWildcards: false
```

or

```
'this is a string' findString: 'string' startingAt: 4  ignoreCase: true useWildcards: false
```

Beck's Rule

When a keyword message has two or more keywords

- Place each keyword with its argument on its own line
- Indent the keyword one tab from the receiver

Program formatting is a matter of personal preference. Some Smalltalk style guides state that keyword messages with two keywords should be placed on one line. Whichever style one uses consistency is very important. Consistent style makes it easier for others to read your code. When you work on a team, the entire team should use the same style. Many companies have programming styles fro all programmers to fallow.

The Tab Verses Spaces Debate

When one indents a line of code do you use:

- Tab

  Easier to type

  Sometimes tabs are different on screen and on hard copy

  Some companies ban tabs

- Spaces

Smalltalk handles tabs uniformly

Use tabs to indent in Smalltalk

Do not use spaces to indent in Smalltalk

Precedence

- First unary messages are parsed left to right
- Binary messages are parsed left to right after unary messages
- Keyword messages are parsed after binary messages

Parenthesis change the order of evaluation

Expression	Result
3 + 4 * 2	14
3 + (4 * 2)	11
5 + 3 factorial	11
(5 + 3) factorial	40320
'12' asNumber + 2	14

Arithmetical operations do not use normal mathematical precedence rules

Parenthesis must be used to separate multiple keyword messages in one statement

```
'this is a string' reversed
    findString: ('the cat is white' copyFrom: 9 to: 10)
    startingAt: 1 + 2
    caseSensitive: 2 + 2 = 4
```

Transcript

Special output window
Similar in purpose to Java's System.out and C++'s out

Useful Transcript messages:

clear
clear the Transcript

show: aString
display aString in the Transcript

print: anObject
display a string representation of anObject in the Transcript

nextPutAll: aString
add aString to the display buffer

endEntry
put contents of display buffer in Transcript
empty the buffer

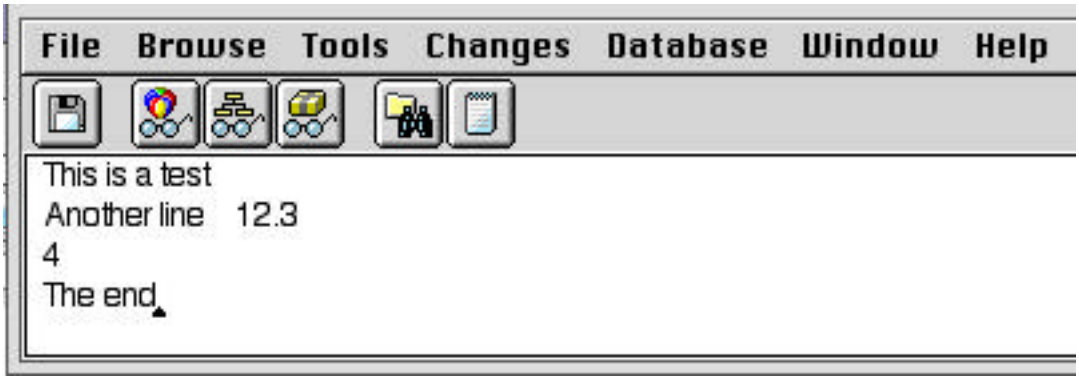
flush
Same as endEntry

tab cr space crtab crtab: anInteger
put given character in the display buffer

Sample Program

```
Transcript clear.
Transcript show: 'This is a test'.
Transcript cr.
Transcript show: 'Another line'.
Transcript tab.
Transcript print: 12.3.
Transcript cr.
Transcript show: 4 printString.
Transcript cr.
Transcript show: 'The end'.
```

Result of Running Program



Cascading Messages

Format:

receiver selector1 [arg] ; selector2 [arg] ; ...

A cascade sends multiple messages to the same receiver

Messages are sent from left to right to the same receiver

```
Transcript
        clear;
        show: 'This is a test';
        cr;
        show: 'Another line';
        tab;
        print: 12.3;
        cr;
        show: 4 printString;
        cr;
        show: 'The end'.
```

Cascade Versus Compound Messages

Expression	Result
'hi mom' reversed asUppercase	'MOM IH'
'hi mom' reversed; asUppercase	'HI MOM'

Compound

In a compound message each message is sent to the result of the previous message

'hi mom' reversed asUppercase

First send reversed to 'hi mom'

The result is 'mom ih'

Now send asUppercase to 'mom ih'

Cascade

In a cascade message each message is sent to the same receiver

'hi mom' reversed; asUppercase

First send reversed to 'hi mom'

The result is not used

Now send asUppercase to 'hi mom'