FREE eBook

LEARNING smalltalk

Free unaffiliated eBook created from
Stack Overflow contributors.

#smalltalk

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: smalltalk

It is an unofficial and free smalltalk ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official smalltalk.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with smalltalk

Remarks

This section provides an overview of what smalltalk is, and why a developer might want to use it.

It should also mention any large subjects within smalltalk, and link out to the related topics. Since the Documentation for Smalltalk is new, you may need to create initial versions of those related topics.

Examples

Installation or Setup

The name Smalltalk usually refers to ANSI Smalltalk or Smalltalk 80 (of which the first is based on). While most implementations are close to the standard, they vary in different aspects (usually referred to as *dialects*).

Each implementation has it's own method of installation.

Well known FOSS implementations are:

Pharo Started as a Squeak fork. (Windows/Linux/Mac OSX). Pharo has its own documentation entry at Stackoverflow Documentation, so please take a look there

Squeak (Windows/Linux/Mac OSX)

GNU Smalltalk (Windows/Linux/Mac OSX)

Dolphin Smalltalk Originally commercial, now free open source. (Windows only)

Cuis Smalltalk A Squeak fork with a focus on reducing system complexity.

Commercial Smalltalks include:

VisualWorks/Cincom Smalltalk Free trial available.

VisualAge Smalltalk Originally by IBM, now Instatiations. Free trial available

Smalltalk/x (Free for personal use?)

GemStone/s Free community edition available.

Other Smalltalk dialects

Amber Smalltalk A Smalltalk that lives in the browser.

Redline Smalltalk Smalltalk for the JVM.

List of Smalltalk implementations on world.st

Hello World in Smalltalk

```
Transcript show: 'Hello World!'.
```

This will print `Hello World!` to the Transcript window in Smalltalk. `Transcript` is the class that allows you to print to the Transcript window by sending the message `show:` to that object. The colon indicates that this message requires a parameter which is in this case a string. Strings are represented by single quotes and single quotes only since double quotes are reserved for comments in Smalltalk.

Read Getting started with smalltalk online: https://riptutorial.com/smalltalk/topic/5316/getting-started-with-smalltalk

Chapter 2: Smalltalk Syntax

Examples

Literals and comments

Coments

```
"Comments are enclosed in double quotes. BEWARE: This is NOT a string!"
"They can span
 multiple lines."
```

Strings

```
'Strings are enclosed in sigle quotes.'
'Single quotes are escaped with a single quote, like this: ''.'
''''  "<--This string contains one single quote"

'Strings too can span
multiple lines'

''   "<--An empty string."
```

Symbols

```
#thiIsaSymbol  "Symbols are interned strings, used for method and variable names,
                and as values with fast equality checking."
#'hello world' "A symbol with a space in it"
#''            "An empty symbol, not very useful"
#+
#1             "Not the integer 1"
```

Characters

```
$a     "Characters are not strings. They are preceded by a $."
$A     "An uppercase character"
$      "The spacecharacter!"
$→     "An unicode character"
$1     "Not to be confused with the number 1"
```

Numbers

Numbers come in all varieties:

Integers:

- Decimal

  10

  -1

  0

  1000000000000000000000000000000000000000000000

- Hexadecimal

  16rAB1F

  16r0

  -16rFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF

- ScaledDecimal

  17s0

  3.14159265s8

- Other

  8r7731 "octal"

  2r1001 "binary"

  10r99987 "decimal again!"

Floating point

```
3.14 1.2e3     "2 floating-point numbers"
```

Fractions

Fractions are not the same as floating-point numbers, they're exact numbers (no rounding error).

```
4/3            "The fraction 4/3"
355/113        "A rational approximation to pi"
```

Arrays

```
#(#abc 123)    "A literal array with the symbol #abc and the number 123"
```

Byte Arrays

```
#[1 2 3 4]       "separators are blank"
#[]              "empty ByteArray"
#[0 0 0 0 255]   "length is arbitrary"
```

Dynamic arrays

Dynamic arrays are built from expressions. Each expression inside the braces evaluates to a different value in the constructed array.

```
{self foo. 3 + 2. i * 3}   "A dynamic array built from 3 expressions"
```

Blocks

```
[ :p | p asString ] "A code block with a parameter p.
                     Blocks are the same as lambdas in other languages"
```

Some notes:

Note that literal arrays use any kind and number of blanks as separators

```
#(256 16rAB1F 3.14s2 2r1001 $A #this)
"is the same as:"
#(256
  16rAB1F
  3.14s2
  2r1001
  $A #this)
```

Note also that you can compose literals

```
#[255 16rFF 8r377 2r11111111]       (four times 255)

#(#[1 2 3] #('string' #symbol))     (arrays of arrays)
```

There is some "tolerance" to relaxed notation

```
#(symbol) = #(#symbol)              (missing # => symbol)

#('string' ($a 'a'))                (missing # => array)
```

But not here:

```
#([1 2 3]) ~= #(#[1 2 3])           (missing # => misinterpreted)
```

However

```
#(true nil false)                   (pseudo variables ok)
```

But not here!

```
#(self) = #(#self)                  (missing # => symbol)
```

As you can see there are a couple of inconsistencies:

- While pseudo variables `true`, `false` and `nil` are accepted as literals inside arrays, the pseudo variables `self` and `super` are interpreted as **Symbols** (using the more general rule for unqualified strings.)
- While it is not mandatory to write `#(` for starting a nested array in an array and the parenthesis suffices, it is mandatory to write `#[` for starting a nested `ByteArray`.

Some of this was taken from:

http://stackoverflow.com/a/37823203/4309858

Message sending

In Smalltalk almost everything you do is *sending messages* to objects (referred as calling methods in other languages). There are three types of messages:

Unary messages:

```
#(1 2 3) size
"This sends the #size message to the #(1 2 3) array.
 #size is a unary message, because it takes no arguments."
```

Binary messages:

```
1 + 2
"This sends the #+ message and 2 as an argument to the object 1.
 #+ is a binary message because it takes one argument (2)
 and it's composed of one or two symbol characters"
```

Keyword messages:

```
'Smalltalk' allButFirst: 5.
"This sends #allButFirst: with argument 5 to the string 'Smalltalk',
 resulting in the new string 'talk'"

3 to: 10 by: 2.
```

```
"This one sends the single message #to:by:, which takes two parameters (10 and 2)
 to the number 3.
 The result is a collection with 3, 5, 7, and 9."
```

Multiple messages in a statement are evaluated by the order of precedence

```
unary > binary > keyword
```

and left to right.

```
1 + 2 * 3 " equals 9, because it evaluates left to right"
1 + (2 * 3) " but you can use parenthesis"

1 to:   #(a b c d) size   by:   5 - 4
 "is the same as:"
1 to: ( #(a b c d) size ) by: ( 5 - 4 )
```

If you want to send many messages to the same object, you can use the cascading operator ; (semicolon):

```
OrderedCollection new
  add: #abc;
  add: #def;
  add: #ghi;
  yourself.
```

"This first sends the message #new to the class OrderedCollection (#new is just a message, not an operator). It results in a new OrderedCollection. It then sends the new collection three times the message #add (with different arguments), and the message yourself."

Classes and methods

Classes and methods are usually defined in the Smalltalk IDE.

Classes

A class definition looks something like this in the browser:

```
XMLTokenizer subclass: #XMLParser
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'XML-Parser'
```

This is actually the message the browser will send for you to create a new class in the system. (In this case it's `#subclass:instanceVariableNames:classVariableNames:poolDictionaries:category:`, but there are others that also make new classes).

The first line shows which class you are subclassing (in this case it's XMLTokenizer) and the

name the new subclass will have (#XMLParser).

The next three lines are used to define the variables the class and it's instances will have.

Methods

Methods look like this in the browser:

```
aKeywordMethodWith: firstArgument and: secondArgument
    "Do something with an argument and return the result."

    ^firstArgument doSomethingWith: secondArgument
```

The `^`(caret) is the return operator.

```
** anInteger
    "Raise me to anInteger"
    | temp1 temp2 |

    temp1 := 1.
    temp2 := 1.
    1 to: anInteger do: [ :i | temp1 := temp1 * self + temp2 - i ].
    ^temp1
```

this is not the right way to do exponentiation, but it shows a *binary message* definition (they're defined like any other message) and some *method temporary variables* (or method temporaries, *temp1* and *temp2*) plus a block argument (*i*).

Loops in Smalltalk

For this example, an `Ordered Collection` will be used to show the different messages that can be sent to an `OrderedCollection` object to loop over the elements.
The code below will instantiate an empty `OrderedCollection` using the message `new` and then populate it with 4 numbers using the message `add:`

```
anOrderedCollection := OrderedCollection new.
anOrderedCollection add: 1; add: 2; add: 3; add: 4.
```

All of these messages will take a block as a parameter that will be evaluated for each of the elements inside the collection.

1. `do:`
   This is the basic enumeration message. For example, if we want to print each element in the collection we can achieve that as such:

   ```
   anOrderedCollection do:[:each | Transcript show: each]. "Prints --> 1234"
   ```

   Each of the elements inside the collection will be defined as the user wishes using this

syntax: `:each` This `do:` loop will print every element in the collection to the `Transcript` window.

2. `collect:`
   The `collect:` message allows you to do something for each item in the collection and puts the result of your action in a new collection
   For example, if we wanted to multiply each element in our collection by 2 and add it to a new collection we can use the `collect:` message as such:

```
evenCollection := anOrderedCollection collect:[:each | each*2]. "#(2 4 6 8)"
```

3. `select:`
   The `select:` message allows you to create a sub-collection where items from the original collection are selected based on some condition being true for them. For example, if we wanted to create a new collection of odd numbers from our collection, we can use the `select:` message as such:

```
oddCollection := anOrderedCollection select:[:each | each odd].
```

   Since `each odd` returns a Boolean, only the elements that make the Boolean return true will be added to `oddCollection` which will have `#(1 3)`.

4. `reject:`
   This message works opposite to `select:` and rejects any elements that make the Boolean return `true`. Or, in other words it will select any elements that make the Boolean return `false`. For example if we wanted to build the same `oddCollection` like the previous example. We can use `reject:` as such:

```
oddCollection := anOrderedCollection reject:[:each | each even].
```

   `oddCollection` will again have `#(1 3)` as its elements.

These are the four basic enumeration techniques in Smalltalk. However, feel free to browse the `Collections` class for more messages that may be implemented.

Read Smalltalk Syntax online: https://riptutorial.com/smalltalk/topic/5422/smalltalk-syntax

Credits

S. No	Chapters	Contributors
1	Getting started with smalltalk	Bananeweizen, Community, fede s., Leandro Caniglia, Stephan Eggermont, ybce
2	Smalltalk Syntax	aka.nice, fede s., ybce