Learning Smalltalk

The best way to learn Smalltalk is by example. I’ll guide you in this tutorial, but feel free to experiment and explore on your own. That’s part of the fun.

# Launching Beagle Smalltalk

Windows:

Launch StartupBeagleSmalltalk.bat

Linux:

Open a shell, CD to the directory containing Beagle Smalltalk and run:

./beagle beagle.im

Then, open BeagleUI.html in a web browser.

After launching, you should see something like this:

A screenshot of a computer

AI-generated content may be incorrect.

If you see “Connected!!” then Beagle Smalltalk is running.

# Open a Workspace

Click on the Workspace button. This will open a new window in the web browser with the title Workspace. This window is a text editor that allows you to run Smalltalk code.

Click in the workspace and type 3 + 4. Select that text with your mouse and press the right mouse button to pop up a menu. From the menu, select Print It.

A blue and white rectangle with black lines

AI-generated content may be incorrect.

If you did this right, you’ll see the answer printed into the workspace.

A blue and white rectangular object

AI-generated content may be incorrect.

The answer is 7. Congratulations, you just ran your first piece of Smalltalk code.

# Messages

Now, let’s introduce some terms to help explain how this works. Smalltalk is called an Object Oriented language. In fact, the people who developed Smalltalk were the ones who coined the term “Object Oriented” to describe what Smalltalk did.

All information in Smalltalk is represented by objects. In our 3 + 4 example, the 3 is an object and the 4 is an object.

You can get things to happen in Smalltalk by sending messages to objects. In our example, we’re sending a message to 3. A message consists of a name and possibly some parameters. In our example, the message is “+ 4”. The name of the message is + and it has one parameter which is 4.

So, you can say that 3 + 4 is computed by sending the message “+ 4” to the object 3. Every message returns an answer. In this case, the answer is 7 which is printed in the workspace.

Let’s try another example. In the workspace, type in “3 squared”. Select it and choose “Print It” from the pop up menu.

A blue and white rectangular object

AI-generated content may be incorrect.

Here’s an example of a message that has no parameters. We send the message “squared” to the object 3. It gives us back the answer 9.

The message “squared” is called a unary message. Unary messages are words that start with a letter (usually a lowercase letter) and consists of a sequence of letters and numbers.

When we send a message to an object, we call the object that we send the message to the receiver. For 3 + 4, the receiver is 3 and the message is + 4. For 3 squared, the receiver is 3 and the message is squared.

We’ve now seen two kinds of messages. Unary messages take no parameters. They’re called unary because there’s only one object involved – the receiver of the message, in our case, the 3. Binary messages are single or multiple symbolic characters such as + \* / and >=. Each binary message works with two objects – the receiver (3) and the parameter (4).

Programmer’s Note:

If you’ve programmed in other object oriented languages, messages in Smalltalk are just like method calls in other languages. In those languages, they normally take the form:

receiver.message(parameters)

If the compiler could handle it (most languages won’t allow it), 3 + 4 would be like saying:

3.plus(4)

But instead of plus, we can use +.

3.+(4)

Similarly, 3 squared is like saying

3.squared()

What do you do if you want to provide more than two parameters? What if your operation requires three objects? Suppose you want to ask if a number is between two other numbers – for example, “is 3 between 1 and 5”?

We can use something called keyword messages to do this. Here’s what it looks like:

3 between: 1 and: 5

A blue and white rectangular frame

AI-generated content may be incorrect.

The words between: and and: are called keywords. Keywords always end with a colon. Following each keyword, we provide an object for that parameter. If the message has two keywords, then we provide two parameters.

In this case, the number 3 is between 1 and 5. The result is true.

The receiver of the message is 3 and the message is between:and: with the parameters 1 and 5. Using keywords, we can provide an explanation of each parameter.

Programmer’s Note:

This example would be like writing:  
3.betweenAnd(1, 5)

Other object oriented languages usually don’t name the methods to correspond to the number of parameters.

Let’s try some other examples of messages.

3 \* 4 ==> 12

Multiplication uses the \* symbol

3 \* 4 – 2 ==> 10

Calculate 3 + 4 to give 12. Then use the 12 as the receiver and send – 2 to get 10

12 – 2 \* 4 ==> 40

We don’t do order of operations. We go left to right.  
12 – 2 ==> 10  
10 \* 4 ==> 40

12 – (2 \* 4) ==> 4

We can use parentheses () to force the order  
2 \* 4 ==> 8  
12 – 8 ==> 4

3 squared ==> 9

We can use squared to multiply the number by itself

3 + 5 squared ==> 28

Always send unary messages before binary messages  
 5 squared ==> 25  
 3 + 25 ==> 28

3 + 4 between: 2 squared + 1 and: 3 + 5 squared ==> true

Keyword messages are always sent last  
 The calculation is done in the following order  
 3 + 4 ==> 7  
 2 squared ==> 4  
 4 + 1 ==> 5  
 5 squared ==> 25  
 3 + 25 ==> 28  
 7 between: 5 and: 28 ==> true

That last one may take some explaining. To read this, you always go left to right. As you go, you need to choose whether to send the message to the left or to the right of the current item. Always choose unary over binary over keyword.

Here’s how you can think about that last calculation:

3 + 4 between: 2 squared + 1 and: 3 + 5 squared  
7 between: 2 squared + 1 and: 3 + 5 squared  
7 between: 4 + 1 and: 3 + 5 squared  
7 between: 5 and: 3 + 5 squared  
7 between: 5 and: 3 + 25  
7 between: 5 and: 28 ==> true

This takes some getting used to. If it gets confusing, just use () brackets to specify the order.

# Classes

We explained earlier that 3 is an object. Every object belongs to a class. We say that an object is an instance of a class. We can ask any object what its class is.

3 class ==> SmallInteger

So, the class of 3 is SmallInteger. Why is it a “small” integer? Let’s try using a message called factorial. 5 factorial is 5 \* 4 \* 3 \* 2 \* 1.

5 factorial ==> 120

Now, let’s try 100 factorial. This is 100 \* 99 \* 98 … \* 3 \* 2 \* 1.

100 factorial ==> 933262154439441526816992388562667004907159682  
64381621468592963895217599993229915608941463976156518286253  
697920827223758251185210916864000000000000000000000000

This is a pretty big number. Let’s ask that for its class

100 factorial class ==> LargePositiveInteger

It turns out that any integer larger than 1152921504606846975 will become a large integer.

Programmer’s Note

Unlike most other programming languages, Smalltalk doesn’t have a limit on the size of Integers. If the number gets too large, Smalltalk automatically switches to large integers which have an arbitrarily large size.

To help understand how classes work, let's try a different example.

3 @ 4 ==> 3 @ 4

The @ message creates a Point object that prints itself as "<x> @ <y>". Every Point contains an x and a y. The expression 3 @ 4 creates a Point with 3 for the x and 4 for the y, so it prints itself as 3 @ 4.

Try running 3 @ 4 with the Inspect It menu item. It runs the selected code and opens a window called an inspector to show the result.

A screenshot of a computer

AI-generated content may be incorrect.

The title bar shows that this object is an instance of the class Point. If you click on self, you can see how the point prints itself. Click on x or y to see the values of x or y. These are called instance variables. Instance variables are like little notepads where you can write one object. In the instance variable called x we've written 3 and in the instance variable called y we've written 4. We'll talk more about variables later. The - r and - theta are called derived values. They show you the results of calculations on this object. In this case, r is the length of the line from 0 @ 0 to this point, and theta is the angle from that line to the X axis.

1

1

2

3

r

theta

0 @ 0

3 @ 4

2

3

4

x

y

You can close the inspector window now.

Back in the workspace, try this:

(3 @ 4) r ==> 5.0

The message r is a unary message for points that calculates the length of the line from 0 @ 0 to the point. Notice that the answer is printed as 5.0. This is a Float. Floats are numbers that can have a decimal point and numbers after the decimal. It’s unusual for the value of r to be an integer, so the r message always returns a Float.

When we send the message r to a Point, Smalltalk needs to find the code to run. We call the code that runs in response to a message a method. Let's find the method for r. Click on the Implementers... button on the desktop and type r then hit OK. This opens a new window which shows that r is implemented in the class Point and shows the Smalltalk code for that method.

A screenshot of a computer

AI-generated content may be incorrect.

r

^(self x squared + self y squared) sqrt

There are several things to explain here. Let’s start with the first line which is the name of the method. Since r is a unary method, we don't expect any parameters afterward.

The ^ character means that we need to return from the method with the result of the following expression as the return value. All methods return something. Without a ^, the method will return the object that received the message (which we call the receiver). Within a method, we can refer to the receiver with the word self.

Now, let’s see how this method works when we run (3 @ 4) r. The receiver is the Point 3 @ 4. Smalltalk finds the method called r for the class Point and runs it using 3 @ 4 as self. The self x sends the message x to the Point 3 @ 4 which locates the method x for the class Point (try finding that method) and runs it to return the 3. The next message is squared which is sent to 3 and returns 9. The self y returns 4 and squared returns 16. Then, 9 is sent + 16 which returns 25. Finally, 25 is sent sqrt which returns 5.0. So, the r method returns 5.0.

Let's summarize what we've learned so far.

* Information is represented by objects
* Each object is an instance of a class
* You can send messages to objects
* There are unary, binary, and keyword messages
* When you send a message to an object, Smalltalk looks for the method with that name for that class and calls it.
* The object that the message was sent to is called the receiver
* Within the method, the word self refers to the receiver
* Every method returns an object
* By default, a method will return self (the receiver)
* You can return a different object using the ^ character

# Class Hierarchies

Let’s look back at a message we learned about in the previous section.

3 squared

Since we can send squared to 3, there must be a method to run. We know that the class of 3 is SmallInteger:

3 class ==> SmallInteger

Let’s look for Implementers… of squared.

A screenshot of a computer

AI-generated content may be incorrect.

There’s a method called squared but it’s in a class called Number, not SmallInteger. How does that work? Let’s look at the class Number in a System Browser. Click the button called System Browser.

A screenshot of a computer

AI-generated content may be incorrect.

This tool is called the System Browser and is the main tool we use to view and change Smalltalk code. It has four lists across the top and a text editor below. The first list shows Kits. A Kit helps organize code into large categories. Kits contain classes. Let’s find the class called Number. From the kit list, popup a menu and select “Find Class”.

A screenshot of a computer

AI-generated content may be incorrect.

In the popup window, enter Number and press Ok.

A screenshot of a computer screen

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

The browser found the class Number in the Kit called Core. The second list of a list of classes in the kit. The third list shows the protocols of the methods. We’ll see that in a few minutes. From that list, though, pop up a menu and select Find Method. Type in squared and press Ok.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer screen

AI-generated content may be incorrect.

The browser found the method squared in the protocol called arithmetic.

A screenshot of a computer

AI-generated content may be incorrect.

Now, go to the second list (the Class list) and from the popup menu, select Show Class Hierarchy.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

Classes are organized into hierarchies. The class called Number has subclasses called Integer, LargePrecisionReal, and Fraction. In turn, the class Integer has subclasses called SmallInteger and LargeInteger. Any methods defined in a superclass (like Number) are accessible to all subclasses (such as SmallInteger).

When we send squared to the object 3, Smalltalk looks for the method in the class SmallInteger. It’s not there, so Smalltalk looks in the superclass Integer. It’s not there either, so Smalltalk looks in the class Number. It finds the method there and runs that method on 3.

squared

^self \* self

Since the receiver is 3, self means 3 and self \* self is 3 \* 3 which produces 9. The squared method then returns the answer 9.

# Variables

Let’s look at the following code.

| message |

message := 'hello'.

message size

The object 'hello' is called a String. A string is a collection of characters. The word message refers to a variable. Imagine a variable as a note that has a name and space to write an object. The vertical bars declare that we have a variable called message. The := assigns the result of the expression to the right into the variable. Finally, **message size** tells Smalltalk to send the message **size** to the object stored in **message**.

Programmer’s note:

In Smalltalk, we don’t declare the type of the variable. In other languages, the type restricts the kinds of objects that can be stored in a variable and plays an important part of the message sending mechanism. This is called static typing (or, more precisely, manifest typing). Static types introduce restrictions. Smalltalk has no such restrictions. We can argue about whether that’s a good thing or a bad thing, but in Smalltalk, you can send any message to any object. If that object has a method with that name, it will run the method.

Note that a variable doesn’t “contain” an object but rather “refers to” an object.

message

This means that multiple variables can refer to the same object.

| message1 message2 |

message1 := 'hello'.

message2 := message1.

message1 size

message1

message2

Variables that are declared between vertical bars are called temporary variables. Temporary variables will disappear once the code containing these variables has finished running and has returned.

As we saw earlier, objects can contain variables. The point 3 @ 4 has variables for x and y. These are called instance variables.

The other kind of variable is called a Global variable. Global variables are visible by any Smalltalk code. As it turns out, all classes are stored in global variables. Let's try this.

Point x: 3 y: 4

This is another way of creating a point like 3 @ 4. (Try finding the implementors of @) In this example, Point is a global variable pointing to an object that represents the class Point.

If you want to create your own global variable, you can use this:

Smalltalk at: #MyVariableName put: nil

The word nil is a special word that refers to an object. We use that object as the default value for variables. It basically means "nothing".

Programmer's Note

The word nil in Smalltalk is like null or NULL in other object oriented languages.

After that, you can use MyVariableName just like other variables.

MyVariableName := 5.

MyVariableName squared ==> 25

Programmer's Note

Other Smalltalk dialects have variables called Class Variables and Pool Variables. Those variables have a very wide scope and are very hard to manage properly. Beagle Smalltalk doesn't include those variables for simplicity.

Notice the period at the end of the first line. Periods in Smalltalk are statement separators. It means that a statement is finished and another is starting.

Programmer's Note

Other C-like languages use ; as statement separators.

# Characters, Strings and Symbols

Let's look at some more basic objects.

Characters are objects that represent single letters, digits, or punctuation marks. You can create one using the $ character.

$a ==> The character a  
$; ==> The character ;

Strings are sequences of characters.

'hello' ==> A sequence of the characters $h $e $l $l $o

If you want to put a single quote into a string, you need to type it twice

'I can''t do that'

A Symbol is very much like a String. It's a sequence of characters. When you create a Symbol, however, we use a symbol table to lookup an existing Symbol with those characters and use that Symbol if it's there instead of creating a new one. This means that two variables that point to symbols with the same letters are guaranteed to point to the same object.

You can create a symbol using a # character followed by a sequence of alphanumeric characters.

#hello25

If you want to include different characters, you can use single quotes like this:

#'symbol with spaces'

# Booleans

Smalltalk has two special words for Booleans – true and false.

3 > 2 ==> true  
 3 > 4 ==> false

You can use true and false as if they were variables.

| myBoolean |  
 myBoolean := true

# Arrays and ByteArrays

Arrays are fixed-sized collections of objects. Here's an example.

#(9 'hello' 5.4 #there)

This creates an array of size 4 containing a mix of integers, strings, floats and symbols.

Programmer's Note

In most object oriented languages, Arrays are restricted to hold only one kind (or type) of object. Smalltalk has no such restrictions.

Once we have an array, we can send messages to it. For example,

#(9 'hello' 5.4 #there) size ==> 4  
 - returns the number of objects in the array

#(9 'hello' 5.4 #there) first ==> 9  
 - returns the first element

#(9 'hello' 5.4 #there) last ==> #there  
 - returns the last element

#(9 'hello' 5.4 #there) at: 3 ==> 5.4  
 - returns the element at index 3

#(9 'hello' 5.4 #there) at: 3 put: 23.3  
 - replaces element 3 with the object 23.3

Programmer's Note

Collections in Smalltalk are generally indexed based on 1 where the first element is at index 1, the second is at index 2, etc. This is different from many other languages where the first element is index 0 and the second element is index 1.

You'll find that most messages for Arrays also work for Strings, Symbols, and many other collections.

ByteArrays are special arrays where the elements are numbers between 0 and 255. ByteArrays are generally used to implement low-level operations and aren't commonly seen in regular code.

#[9 67 215 19]

# Blocks

A block is an object that holds Smalltalk code that can be executed later. Blocks are enclosed in square brackets.

[3 + 4] ==> aBlockClosure

Once created, you can run the code in the block by sending it a value message.

[3 + 4] value ==> 7

Blocks can take parameters. You declare parameters by starting each name with a colon (:) and finishing the parameter list with a vertical bar. You run the block using a value: method with as many value: keywords as you have parameters.

[:x :y | x + y] value: 3 value: 4 ==> 7

Smalltalk doesn't have any control structures like IF or WHILE built into the language. We use regular messages and blocks to do that. For example,

age >= 18 ifTrue: [self doAdultAction]

The message ifTrue: is implemented for True and False:

True  
 ifTrue: aBlock  
 ^aBlock value

False  
 ifTrue: aBlock  
 ^nil

So, if the receiver of ifTrue: is a true object, it runs the block. Otherwise, it does nothing and just returns nil.

You can also use ifFalse: if you want to execute code in the false case, or ifTrue:ifFalse: to provide blocks for both the true and false cases.

age >= 18  
 ifTrue: [self doAdultAction]  
 ifFalse: [self doChildAction]

We can use blocks to create loops that run until some condition is true.

| number factorial |

factorial := 1.

number := 20.

[number > 1] whileTrue: [

factorial := factorial \* number.

number := number – 1].

factorial

In this example, the whileTrue: message is sent to the block [number > 1]. If it returns true, it runs the other block then tries the first block again. It continues this until the first block returns false.

We can loop over a range of numbers using to:do:.

| factorial |

factorial := 1.

2 to: 20 do: [:number | factorial := factorial \* number].

factorial

Using blocks, we can loop over collections.

| factorial |

factorial := 1.

#(1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20) do: [:number |

factorial := factorial \* number].

factorial

# Remaining Syntax

There are only a few more things to understand in order to understand the full syntax of Smalltalk. Comments are delimited by double quotes.

"This is a comment"

In a method, you can use super instead of self to change the method lookup mechanism. Super means "send a message to the same receiver as self but start the lookup in the class above the one where the current method is defined."

initialize

super initialize. "do the initialization from the superclass"

self doMyInitialization

Finally, thisContext is used to create a stack trace for the debugger tool and is seldom used by user code.

# Syntax

The following table shows the entire syntax of Beagle Smalltalk. If you understand the concepts below, you'll be able to read any Smalltalk code. Everything else is in the Smalltalk library.

|  |  |  |  |
| --- | --- | --- | --- |
| **Objects** | | **Messages** | **Misc** |
| **Literals**  **Integers**  4 -6 16r5E  **Floats**  4.0 -6.3e-5  **Strings**  'hello'  'I can''t do that'  **Symbols**  #hello  #'fancy symbol'  **Characters**  $a $0  **Arrays**  #(3 'hello' there)  **ByteArrays**  #[47 198 209]  **Blocks**  [3 + 4]  [:x | x + 5] | **Variables**  **Temporary**  | myVar |  **Instance**  x y  **Global**  MyGlobalVar  **Special words**  self  super  nil  true  false  thisContext | **Unary**  squared  **Binary**  + \* <=  **Keyword**  between:and: | ( )  ^  .  :=  <primitive: 25>  | temps |  ;  "comment" |