LittleSmalltalk – Tutorial For Beginners

Author: Danny Reinhold
Version: 0.1
Status: Published
Date: 20th March 2007

Summary:
LittleSmalltalk is a powerful but yet simple programming language and development environment inspired by the famous object oriented Smalltalk language and environment.
It is a fully usable dialect of the Smalltalk programming language, but does not conform to any Smalltalk standard (neither to ANSI-Smalltalk nor to Smalltalk-80) and it is not intended to be.
This document is a short introduction to the language and the environment and shall be a starting point for beginners that have never seen Smalltalk in general or LittleSmaltalk in special before.

The LittleSmalltalk Project

Smalltalk has been developed in the famous XeroX PARC (Palo Alto Research Center) alongside with some other things well known, like the mouse and the concept of graphical user interfaces. It was the first truely object oriented programming language and is even today a highly powerful lanugage and tool for the advanced software developer as well as for beginners. In fact Smalltalk dialects are often used to teach children.

The LittleSmalltalk system was the first Smalltalk variant developed outside of the XeroX PARC. Timothy Budd wrote the first version in the mid-80ies in C and described it in depth in his famous book „A LittleSmalltalk“. (For references please visit the project web site).
Timothy also published several further versions of the system until version 4. Then he lost interest in LittleSmalltalk and worked on other things (for example he created a LittleSmalltalk like system in Java named SmallWorld).

In 2003 I (Danny Reinhold) have shown interest in the project and asked Timothy if he would allow me to continue the project and to change the original license terms to make the system truely free (version 4 was restricted to non-commercial usage).
Gracefully Timothy agreed.

Well, as often with such enthusiastic projects I was highly busy and didn’t find much time to work on the system. A crash of a hard disk without backups caused me to loose all my LittleSmalltalk related work then.
But in december 2006 I found time again and improved LittleSmalltalk in several ways. I added bindings to the GUI toolkit IUP and to the database management system SQLite, fixed several bugs in the system and hacked a small development environment.
At the end of december 2006 I also found some time to create a new web site for the project and to publish the system.
Soon there was a small community and some contributors provided additional bug fixes and even ported LittleSmalltalk to Linux and Mac OS X.

Today (20th March 2007) the system is already usable and possibly interesting for other people, too. It is far from being ‚ready' or so. But it is usable even for larger projects and makes good progress.

So, well. That is where we are now. Lets start with the more interesting things…

What is so interesting about Smalltalk?

Smalltalk is a so called object oriented language. Many programming languages claim to be object oriented, but there is something different.
In Smalltalk really everything is an object. Programming with Smalltalk means that you take an object (for example a cooking plate) and tell it to do something (for example becoming hot) – you send messages to objects and the objects do something in response.

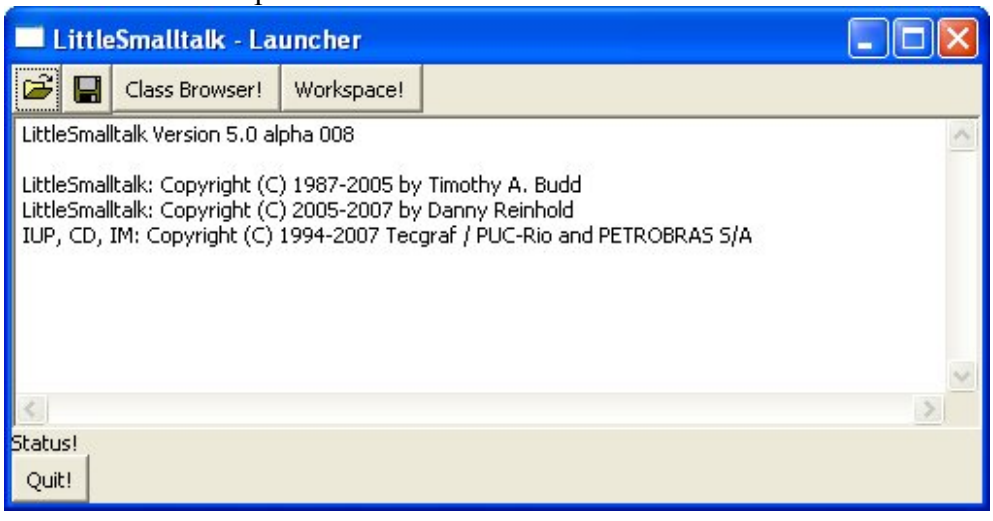
While it is simple to understand that everything in the domain of a problem that a program shall solve is an object (like apples, cooking plates, neighbours, customers etc.) it is not that evident what it means that really everything is an object.
You will understand what this mean later. For now simply accept that ‚object' is an important term when talking about Smalltalk and believe that you will know what an object is after you worked through this tutorial.

Also really important is the fact that you usually do not distinguish between the programming language Smalltalk and the Smalltalk development environment (the tools). You always work with the development tools and not with external text editors or the like when you develop Smalltalk code. This is a strange fact for developers that have worked with other programming languages and than take a look at Smalltalk.
You will immediately start working with the LittleSmalltalk tools and learn what this interactivity is good for.

This paragraph didn't clarify anything and probably leaves more questions than answers. But when you read it again after you finished the tutorial you will see things clear.
Lets start with something real now.

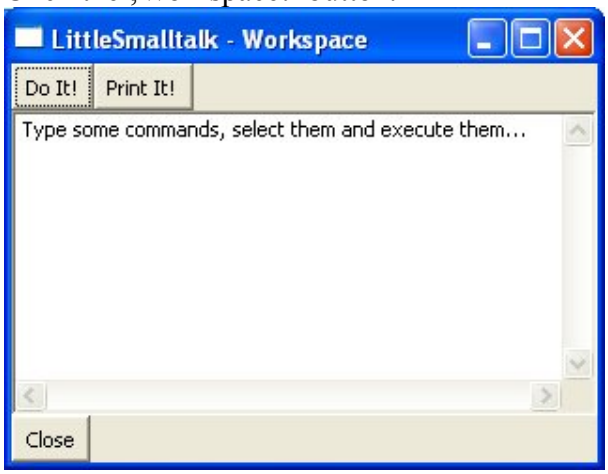
The LittleSmalltalk tools

Download and unzip LittleSmalltalk. Then start the executable.



The start window that you see is the so called ‚Transcript' window. The Window provides some buttons that you can use to start some more specialized LittleSmalltalk tools and a button to exit the LittleSmalltalk system.
But the most important feature of the Transcript window is the text area. This is where output is presented when you for example try to display strings etc.

Click the ‚Workspace!' button.

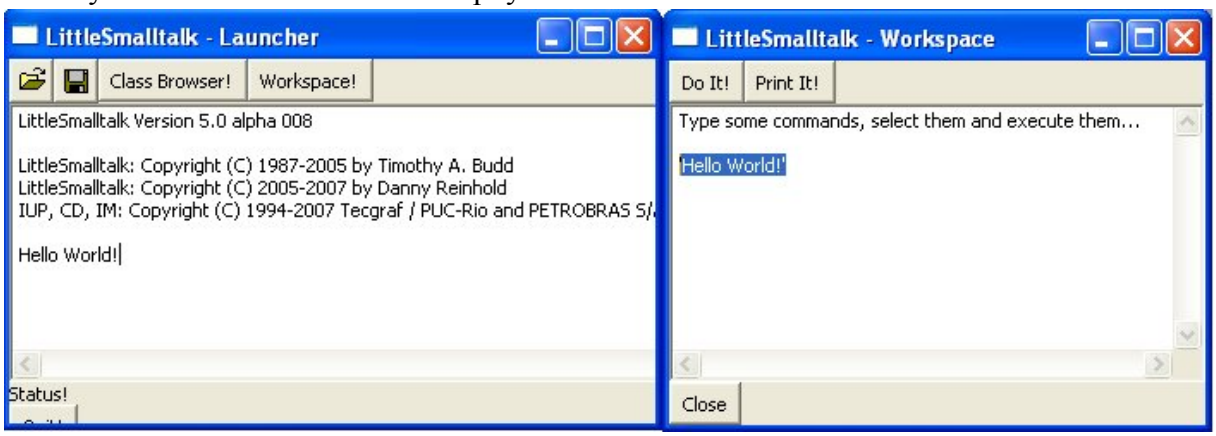


A Workspace window opens which also presents some buttons and a text area. This is where you can type and exectue commands. Whenever you produce Smalltalk code you will probably open a Workspace window and type something to test your Smalltalk code.

Let's try something…
Type this into the text area of the Workspace window:
`'Hello World!'`

Then mark the complete text with the mouse and then click the ‚Print It!' button.

When you then look into the transcript you will see the text `Hello World!`.



What happened here?
You wrote a string into the Workspace are. A string in LittleSmalltalk is a piece of text that is enclosed in single quotes.
Then you marked that string with the mouse. This is neccessary to tell the buttons on which part of the content of the Workspace window an operation shall be performed.

By clicking the button ‚Print It!' you caused LittleSmalltalk to execute the selected text and to print the result of that execution to the Transcript window.

Well, executing a string in LittleSmalltalk does nothing special – and the result of the execution of a string is simply the string itself. So the text that is associated with the string is printed to the Transcript window.

Now write this text to the Workspace window:
```
,Hello World!' print
```

Then select that line, click the ‚Do It!' button and look into the Transcript window. The effect is the same as in the previous example.
The ‚Do It!' button causes LittleSmalltalk to simply execute the selected text.

In other words:
‚Do It!' selects the selected text. ‚Print It!' does the same but then also prints the result of the execution to the Transcript window.

You can verify this if you simply select the line again and click to ‚Print It!'.
Now the string is printed twice! The first time during the execution of the string and then because ‚Print It!' causes the result of the operation to be printed.

Passing a message to an object

You just wrote a piece of object oriented code! Really!
The string `'Hello World!'` is an object.
In Smalltalk each object has a class. The class of an object specifies basically what you can do with objects of that class. Therefore a class contains so called methods.
The object `'Hello World!'` is of class `String`. The class `String` contains a method named `print`.
Writing an object a followed by a method name b means that a message is passed to the object a that a shall execute the method b provided by its class.
So `'Hello World!' print` causes the string `'Hello World!'` to execute the method `print` that is defined in the class `String`.

Arithmetic with LittleSmalltalk

Type `3 + 5` into the Workspace window, select that text and ‚Print It!'.
You will see the result `8` in the Transcript window. This fact is not really interesting on its own. But it is important to understand what happens when LittleSmalltalk generates the result.

`3` is as you may already expect now an object. + is a method in all numeric classes, so you can cause the object 3 to execute the method +. The spcial thing about + is that it is a binary method. `print` in class `String` is a unary method since it doesn't need any further arguments to do something useful. It simply works on the string on which it is executed.

A binary method such as + requires a further argument. In this case we specify that + shall operate on `3` and take the argument `5`.
The result of this operation is the object `8`.

So there is nothing special about arithmetic in LittleSmalltalk. Adding numbers follows the same mechanics as printing values or writing to files etc.
This is really beautyful. The simple concepts ‚Object' and ‚Message sending' are enough to create a general purpose programming language!
But, stop…
This simplicity has a price. Since mathematics and arithmetics are nothing special in LittleSmalltalk the system doesn't know about the typical operator precedence rules that we human persons automatically apply when thinking about mathematical terms.
The developers of the Smalltalk language thought a while about realizing operator precedence mechanisms within the Smalltalk language but then decided not to do so.
Simplicity is a super goal that they didn't want to break because of traditional rules how we typically do something. I think that there decision was great. But as said it has the drawback that you have to think about every mathematical term you write with Smalltalk.

Lets see why I say this. Write and ‚Print It!' this code:
```
5 + 3 * 2
```

A human would read it this way:
5 + (3 * 2)
„Multiply 3 by 2 and add the result to 5."

But the LittleSmalltalk system (as every Smalltalk system) reads it this way:
(5 + 3) * 2
„Add 3 to 5 and multiply the result by 2."

The system is extremly simple and consistent. You have objects and send messages to them. Terms are evaluated strictly from left to right. Simple and good!

Of course LittleSmalltalk offers a way to specify another precedence. You cannot specify a general precedence rule like „evaluate * before +" via operator priorities or such stuff. But you can simply use parenthesis as you also could do in human arithmetics.

You can simply write and execute:
```
5 + (3 * 2)
```
The result is the same that a human reader would expect.

Keep this simple rule in mind:
Always use parenthesis when you write non-trivial arithmetic terms!

Keyword methods

You always know that LittleSmalltalk provides unary methods like the method `print` in the class String and binary methods like + in all the numeric classes.

The difference is that unary methods don't require further arguments and operate solely on the object on which they are called and that binary methods require an additional argument that is written directly behind the operator itself.

There is one further kind of methods that can take an arbitrary number of additional arguments. Those methods are called keyword methods.

Lets look at a simple example.
Execute this code (via „Print It!"):
```
1 to: 5
```

The result in the transcript looks like this:
```
12345
```

The method `to:` is a keyword method. It is sent to the object `1` and takes one argument, in this case the object `5`. The result is an object that represents the closed interval from `1` to `5` and is printed as `12345`.
In this case there is nothing special that makes `to:` different from a usual binary operator method such as `+`.

Now execute this:
```
1 to: 9 by: 2
```

The result is:
```
13579
```

In this case we have a method that takes two additional arguments. The `to:` and `by:` are not separated methods. They form the combined keyword method `to:by:`.
When we call this method we write `to:` followed by the argument that shall be the end of the interval followed by `by:` and the step that we whish between two numbers.

It is important to understand that `to:by:` is really only one method that, when applied is splitted into several parts. This a a great way to write clear code. All arguments to a method are automatically commented because you split the method name up into several parts – each part describing exactly one argument. You will never accidentally switch the `9` and the `2` when you call `to:by:` because you immediately see which argument is meant for what!

Objects and classes

As said above each object has a class where the class specifies what you can do with the object or in other words which methods are understood by the object.

To determine the class of an object you can simply ask it. Execute and print these lines:
```
'Hello World!' class.
3 class.
3333 class.
```

You will see these results:

```
String
SmallInt
LargeInt
```

So the string is of class `String`, `3` is of class `SmallInt` and `3333` is of class `LargeInt`.

A highly interesting fact about Smalltalk is that classes are also objects.
So you can for example also ask a class for its class.

Execute and print these lines one by one and look at the Transcript to see the output:

```
'Hello World!' class
'Hello World!' class class
'Hello World!' class class class
'Hello World!' class class class class
'Hello World!' class class class class class
'Hello World!' class class class class class class
```

You will see these results:

```
String
MetaString
Class
MetaClass
Class
MetaClass
```

So the class of class `String` is `MetaString`. The class of class `MetaString` is `Class` and the class of class `Class` is `MetaClass`. Then we come into a cycle: The class of class `MetaClass` is `Class`.

You don't need to understand this in details now. But keep in mind that also classes are objects and that each class has a class. So the system is highly consistent and only uses a minimum of concepts.

In fact every class `A` has a class named `MetaA`. You will see later what these meta classes are good for.

Which methods are provided by a class?

Since a class is an object you can send messages to it. For example you can ask a class which methods it provides.

Execute this:

```
String listMethods
```

This will cause the class `String` to print all methods that it provides directly to the Transcript window:

```
+
<
=
```

```
asInteger
asString
asSymbol
at:ifAbsent:
at:put:
break:
collect:
copy
doIt
doItAndResume
edit
from:to:
print
printString
reverse
select:
```

So for example strings knows the operator +. You can use it to concatenate two strings:

```
('Hello' + ' World!') print
```

Or you can convert a string that only contains digits into an integral numeric value and do some arithmetic with it:

```
25 + ('325' asInteger)
```

The result is the number 350.

Message precedence rule

Above I said that Smalltalk doesn't provide operator precedence rules. Well, to be true, this is only part of the truth. In fact you cannot specify that a specific method has a higher priority than another one.
But there is a rule that specifies in which order LittleSmalltalk evaluates an expression:

- Method calls are evaluated from left to right
- But binary methods are evaluated before keyword methods are evaluated
- Unary methods are evaluated first

So:

```
25 + ,325' asInteger
```

is equivalent to:

```
25 + ('325' asInteger)
```

```
1 to: 25 by: 5 + '5' asInteger
```

is equivalent to:

```
1 to: 25 by: (5 + ('5' asInteger))
```

This precedence rule makes life simpler and saves lots of parenthesis since unary methods occur really often.

Browsing classes

Lets have a look into some system classes to understand some concepts better.
In the transcript window click the button „Class Browser!“.
A new window gets opened – the so called class browser.
This is the tool that you use most often when working with LittleSmalltalk.

The window is separated into several areas. In the left upper area you see a list of packages that are currently installed. A package contains classes. You can install and uninstall packages to exchange sets of classes between installations of LittleSmalltalk.

The most basic classes are contained in the package „System“. Please click onto „System“ in the package view to make „System“ the current package.

The class list shows all classes that are contained in the current package. Click onto the class „Boolean“ in the class view to make „Boolean“ the active class.

Then you see in the method view (upper right area) all methods that are specified within the active class. These are the methods that are understood by objects of the current class. So when you selected the class Boolean you will see all methods that are understood by boolean objects.
A boolean object represents a truth value. So the methods in class Boolean deal with truth values.

In the dialog's title you see some some information:

- The active package
- The active class
- The parent class of the active class

This is interesting – a parent class… You should know some further details and concepts about classes. As said earlier each object has a class and that class specifies what you can do with objects of that class.
It happens quite often that several classes have many things in common. In those cases it is wise to factor out the common behaviour into another class and make the classes subclasses of that class.

A theoretical example:
We think about two classes „Man“ and „Woman“.
Both classes should have the methods „age“ and „gender“ to return the age resp. the gender of the man or woman object.
While the method „gender“ looks different in both classes (in Man it would simply return ‚male' and in Woman ‚female') the method „age“ would look identically in both classes.
So to not copy'n'paste identical code through our application we specify a third class named „Person“. This class also has the methods „age“ and „gender“.
Now we specify that Woman and Man are subclasses of class Person. This means that we don't need to implement the method „age“ in Woman or in Man since it is already available in the common parent class Person.
And we won't implement „gender“ in Person since the specialized classes Man and Woman already have exact definitions of this method.

The effect is:
Whenever we have an object m of class Man we can ask it for its gender or its age like this:

```
m age
m gender
```

The method gender is implemented directly in class Man and returns ‚male'.
The method age is not found in the class Man. LittleSmalltalk than delegates the method call to the parent class of class Man which is the class Person. And there the method is found and can be executed.
The same is true when we look at an object of class Woman. The method gender returns ‚female' since it is directly implemented in the class Woman and the method age is delegated to the parent class Person.

This concept of subclassing is extremly important in a language like Smalltalk. The concept is also often called generalization because Person is a more general class than Woman and Man or specialization because Woman and Man are more special than Person.

Back to the boolean classes in LittleSmalltalk. You see that there is a class Boolean that represents truth values. If you look into the class view you will also see the classes True and False that are more specialized and represent the truth values true and false. True and False are subclasses of class Boolean.
The LittleSmalltalk system specifies two objects „true" and „false" that are objects of the classes „True" and „False". So if you send a message to the object true LittleSmalltalk first searches in the class True for the method and then in class Boolean (and then in the parent class of class Boolean etc.).

So select the class True and then the method asString.
You see that the large text area in the bottom of the class browser window gets filles. What you see there is the source code of the method that you selected in the method view.

```
asString
  ^'true'
```

If you look into the method asString of class False you'll see this:

```
asString
  ^'true'
```

The first line of a method's source always contains the name of the method and its arguments. asString doesn't have any arguments, so the first line contains only the name of the method. The rest of the source is indeed the source code of the method. This source code specifies what the method should do when executed on an object.
You see a circumflex ^ followed by an object – in this case either the string ‚true' or the string ‚false'. The curmflex means: „Leave the method and return the object that follows the circumflex".
So the method asString in class True returns the string ‚true' and in class False the string ‚false'.

As you can see you can see and in fact modify even the details and internals of the LittleSmalltalk system directly within the system. This means nothing less than that you can study the complete system and extend it on demand!
Since Smalltalk is an extremly simple language the LittleSmalltalk system is pretty short and also simple. Everybody is able to understand the complete system after playing around for a short time. This is a huge advantage of Smalltalk over other languages and development environments!

Use the class browser as often as you can to study details of the system.

How to create a new class?

Up to now you used and studied predefined classes and objects and methods of those classes. But programming with LittleSmalltalk means of course more. You surely want to know how to create your own classes and how to define methods in those classes.

Lets realize the classes Person, Man and Woman as described above.
Start the class browser, select the package System and the class Object.
Then click the button „Subclass“.

In the window that pops up you have to specify some details of the new class.
Specify these data:

```
Package             System
Class Name          Person
Instance Variables  #birthDay #birthMonth #birthYear
Class Variables
```

Click ok. Then you will find your new class in the class view.
Select the class Person.
You will see that there are now three entries in the „Variables“ section. The variable section in a class specifies which variables will be available in every object of that class. The variables of an object represent the internal status of the object. The variables are visible only within the methods of the object’s class.
Everybody has a birth date, specified by a day, a month and a year. We will use our three member variables to hold the birth date of a person.