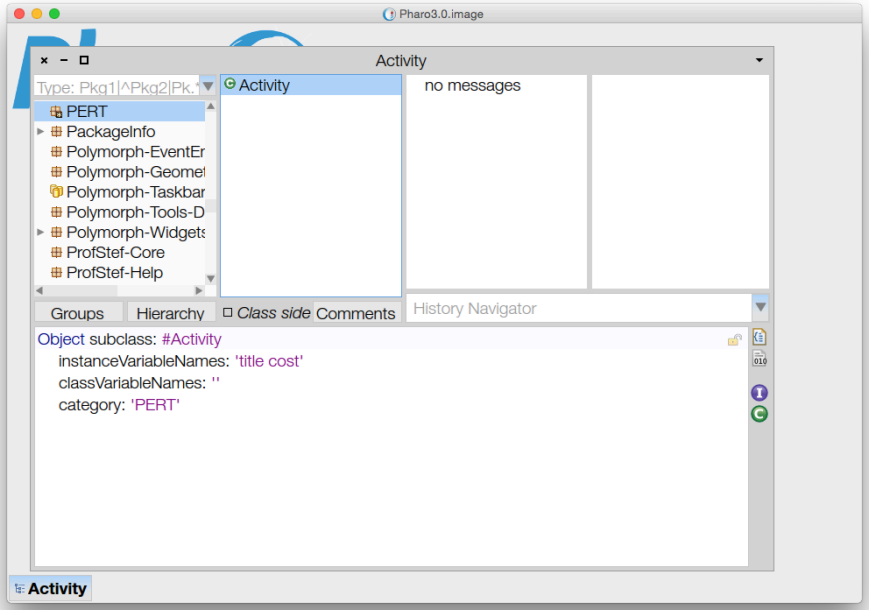
Smalltalk: Working with Collections

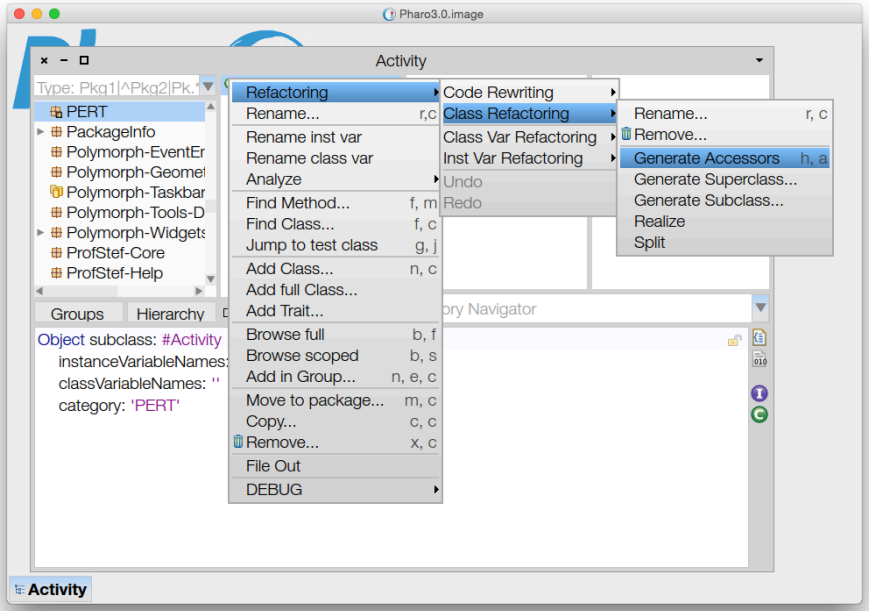
We're able to get away without a lot of explicit looping in Smalltalk because we do a lot work with collections and there are a lot of high-level things you can do with them. What I mean by "high level" is that you're saying what you want rather than explaining to the computer in excruciating detail exactly how to prepare something for you, which is considered "low level."

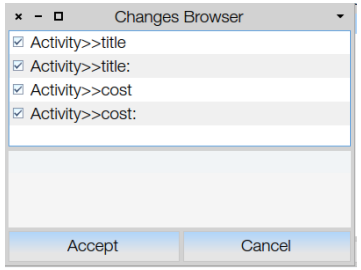
One of the best introductions to this is the collections system, but let's motivate things a little by creating a simple class which we'll use for the PERT algorithm. Let's make PERT tasks! First let's declare the class, and let's keep it completely simple by just having the title of the task and its cost.



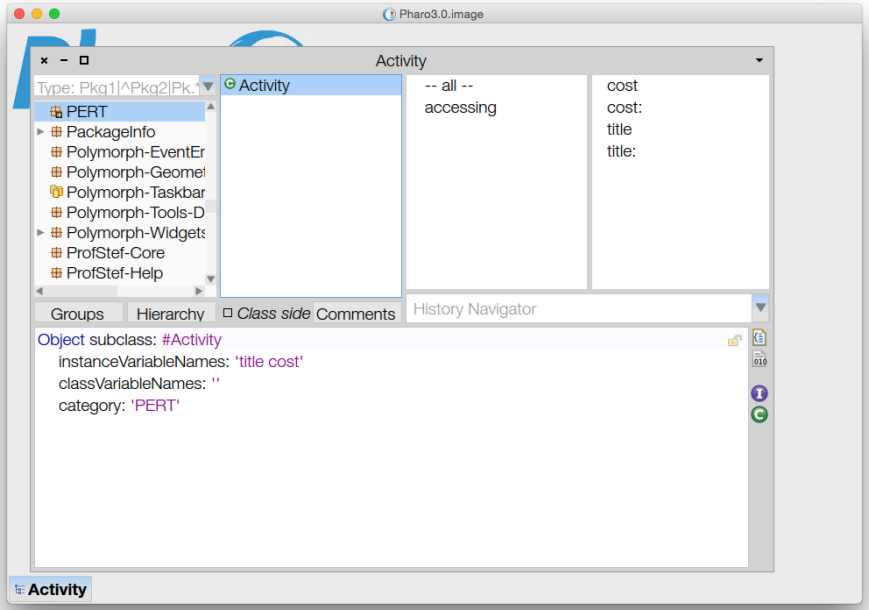
When you open the Browser, just change the text in the lower left box to match this picture and hit Cmd-s to save and create the class.

Now we'll make it very simple by just right-clicking on Activity, selecting Refactoring then Class Refactoring then Generate Accessors. This will bring up a dialog; you can just click Accept, as pictured in these screenshots.



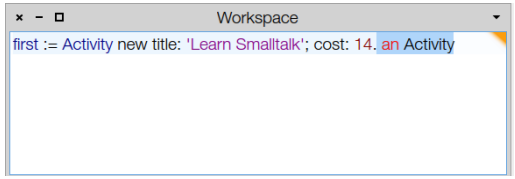


Once these steps are complete the browser should look more like this:

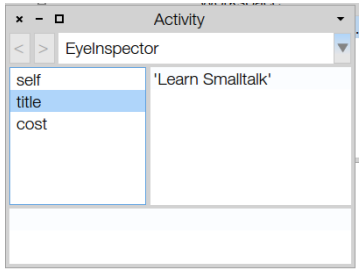


Congratulations, you've just made a very basic class! Notice you now have four methods in the top right box. Your class has to have methods or you won't be able to do anything with it!

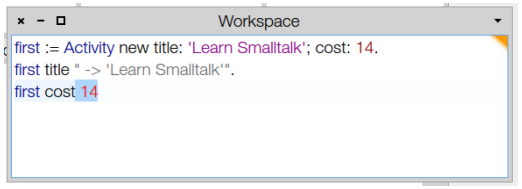
Let's open up a Workspace and noodle with it by making one and seeing what that's like.



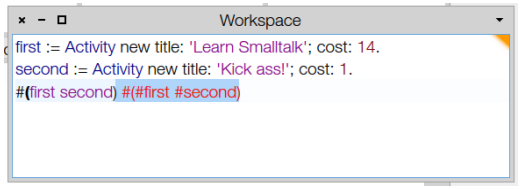
This workspace shows us creating a new Activity. The variable (which belongs to this Workspace) is called `first` and when we did "print it" it gave back "an Activity" because that's the default print representation for an object. If we want to know more about it we can inspect it with Cmd-i, which brings up Inspector:



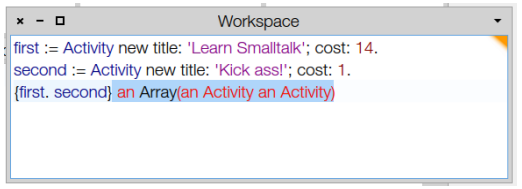
If you look at the properties you'll see this we have a new Activity with the title 'Learn Smalltalk' and the cost 14, just as expected. Close the inspector and let's do a few more print-its with `first`:



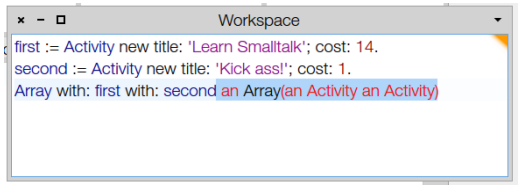
Now let's make a second task and we'll stick them both in an array just for fun.



Oops! This is what's different about those two kinds of literal array. We actually want it to evaluate what's in the list, so we need the other kind of array.



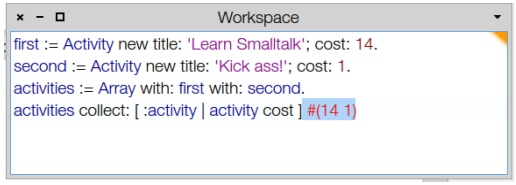
I find this a little weird to look at so let me show you what I would do instead:



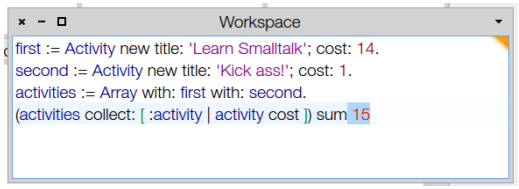
This is just constructing the array with regular message syntax like everything else. Notice we got the same thing this time when we did "print it" as we did on the previous line. This is a matter of personal preference.

Cool, so now we have these arrays. Let's save them in a variable and see what we can do with them. Suppose we wanted to figure out the total cost of all the activities. We could loop through and add up the costs, which would look like this:

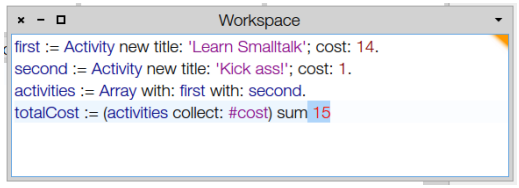
This works, but it's not obvious how. A better way would be one in which the how is a little more obvious. Suppose we first get a list of costs like this:



Then there happens to be a method on Collection for adding up a sum, called sum, which we could use like this:



It turns out that blocks like [:x | x foo] are so common there is a shorthand we can use to simplify the code a little more:

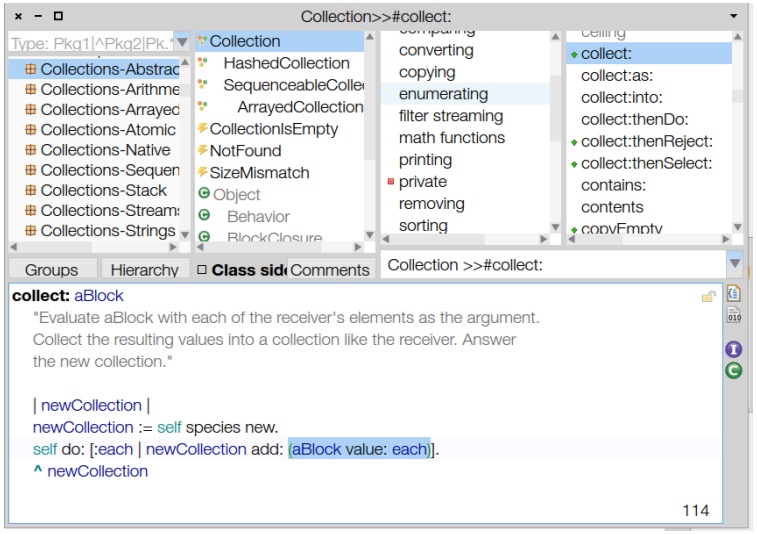


I consider this very high-level code: you can almost read it like English: “the totalCost is the activities costs, summed.”

Unnecessary Detail Section

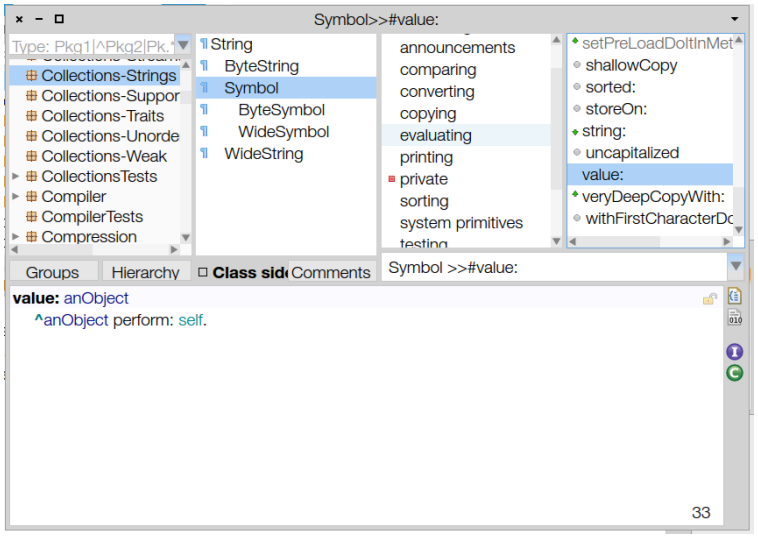
If you are happy knowing *that* this works without caring *why* it works, skip this section.

What we’re doing here is sending the symbol of the cost method to collect instead of a whole block. In fact, we can use this trick whenever a message expects a one parameter block, like collect does. First, let’s look at the code in `collect:` and see what it does. I’ve highlighted the important part:



This is pretty straightforward, it just builds a new object and iterates through each member evaluating the block with each one. I bring it up because the code in `collect:` and related methods doesn’t know what is being passed to it. It could be anything that responds to `value:`. It happens that one-parameter blocks do (this is the BlockClosure class) but anything else that does would also work.

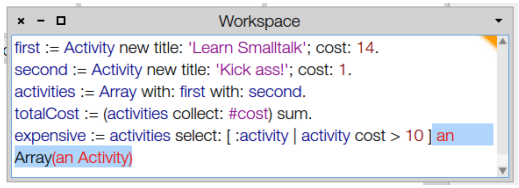
The reason why passing symbols like `#cost` works is subtle but simple. It just has to implement `#value:` to trick `#collect:` and friends, so let’s look at the code in Symbol for `#value:` and see how it works:



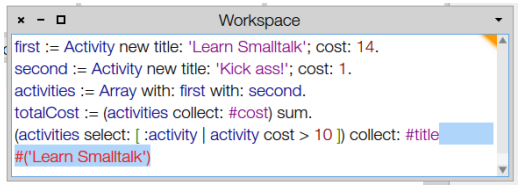
It turns out that `object foo` is the same as `object perform: #foo`. This is great because it means that in Smalltalk we can build messages or store them and send them and it gives us a lot of power and flexibility. This `value:` method on Symbol means that `object foo` is also the same as `#foo value: object` and this is why `activities collect: #cost` works.

Other Collection methods

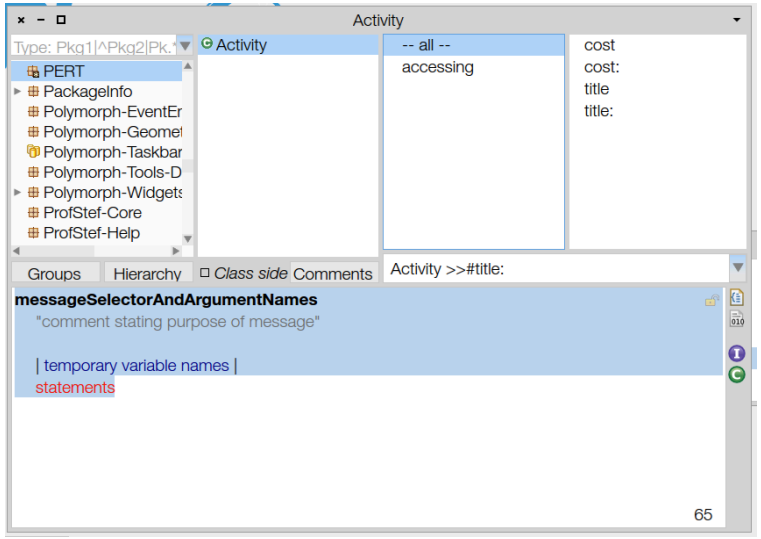
Suppose we just want to see which of our activities are "expensive," which we define as a cost greater than 10, we could do that like this:



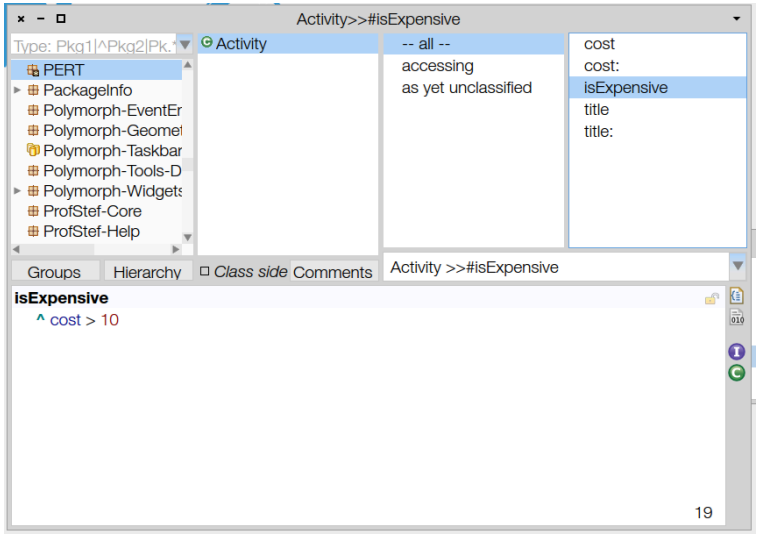
It might be easier to tell which one we're talking about by printing the name afterwards:



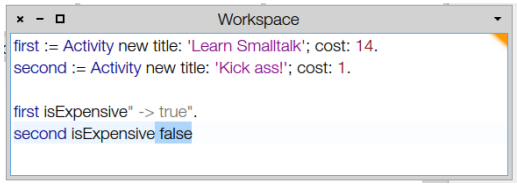
This "expensiveness" idea seems like something the activity might know about itself. Let's move it into a method by going back to the Browser and click on "– all –" in the list of protocols, which will produce this:



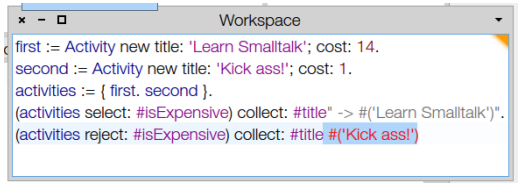
Smalltalk is prompting us to write a method by creating and highlighting a template in the bottom left code panel where it says "messageSelectorAndArgumentNames" and so forth. The rest of the template there is intended to give you something you can edit into being whatever you need. You can also just delete it and type something from scratch, whichever is easier. Let's add `isExpensive` here:



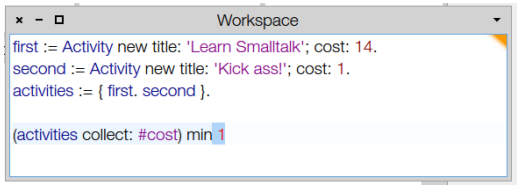
The little caret (`^`) means "return" in Smalltalk. The `cost` is the instance variable we created and otherwise it works just like in any other language. Now we can go back to the workspace and test it out:



Looks like it works. We can use this method with the Collection methods `select:` and `reject:` to find items we're interested in. Suppose we want a list of just the expensive items or just the cheap items:

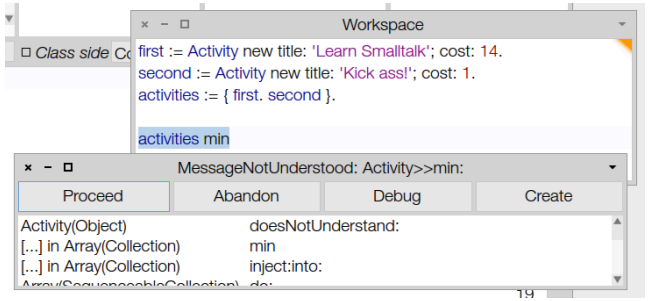


We could also find the minimum cost using `min`:



This is good, but it is building a whole separate list of costs. That seems inefficient, and besides that, what if I want the activity with the lowest cost rather than whatever the lowest cost itself was?

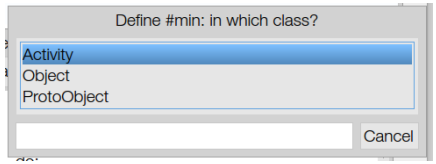
What happens if we just use `min` on the collection of activities itself?



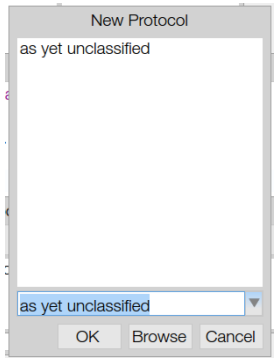
OK, so this is a pretty normal example of something not going right with Smalltalk. We tried a "print it" on this line and we got this Debug window. Whenever this shows up it means something went wrong, and in Smalltalk that usually means somebody didn't understand the message they were sent. In fact, the title of this dialog says it all: Activity doesn't understand `min:` (the `Activity>>min:` syntax means the `min:` message on `Activity`). Notice the difference though: we called `min` on Array, and this is talking about `min:` (with a colon and an argument) on Activity.

The table in this dialog has the object on the left and the message on the right. If the message is inherited, it has the object it is with the object supplying this version of the message in parentheses. So where it says "Array(Collection)" on `min` it means "this is the `min` method called on an Array, but defined in its superclass Collection." We haven't really talked about inheritance yet so if this seems confusing just ignore it.

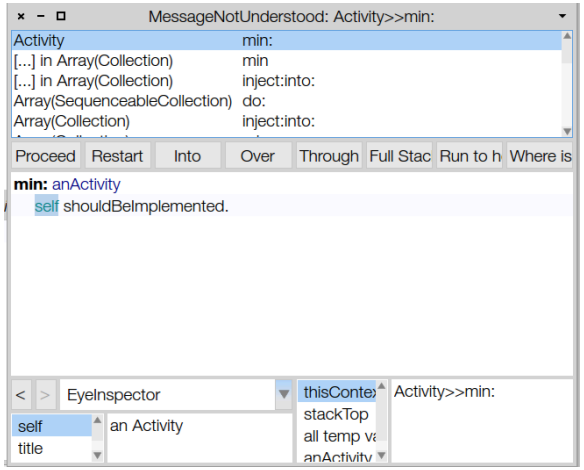
The dialog has some buttons we can hit; most of the time you'll either close it or hit Debug, but in this case you can hit Create instead and we can make the `min:` message on Activity right now. You'll get a series of dialogs afterwards:



Just select Activity. This is the inheritance tree, but we don't want to push this method up the tree any further than Activity.



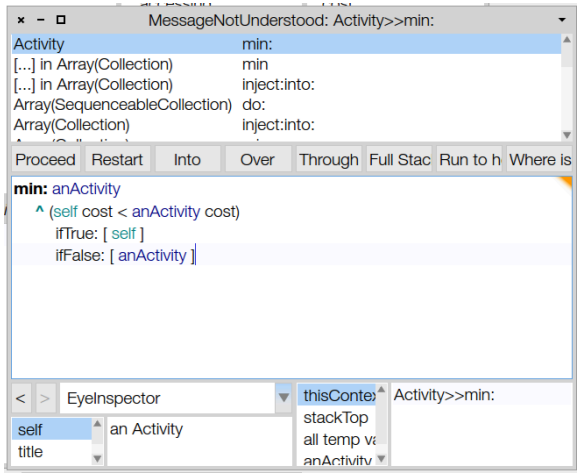
This is just asking for the method to be classified but we can also do this anytime later so just accept "as yet unclassified" for now.



This is a lot like the method template in the Browser it's just inside the debug dialog. The line `self shouldBeImplemented` is an assertion which will cause a Debug window to pop up if it's evaluated, so we're going to delete that code and add our new code.

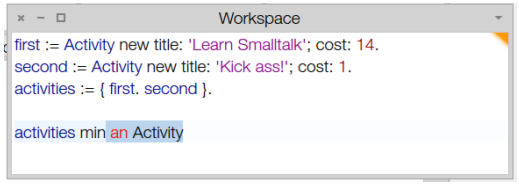
The idea behind `min:` is that it looks at both the receiver and the parameter and decides which one is smaller. For Activities, we should just use the cost for this determination; nobody's going to want the minimum by title length, for instance.

This is a pretty simple conditional. If you were nervous about it you could write it in a Workspace and figure it out, but if it's wrong it's just going to bring you back to the Debug window so there's no real harm in figuring it out here instead.

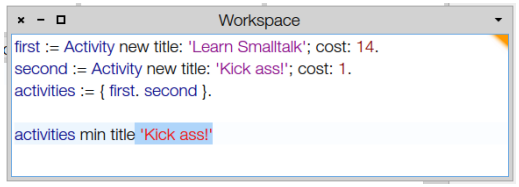


This should be pretty straightforward to follow, except that the return caret can go either inside both blocks or outside the whole test. This looks a little different from C but the same basic ideas are in play: we're just comparing our cost to their cost and returning us or them depending on which one is smaller.

Now that it's implemented, save and hit Proceed.



In a slightly anticlimactic way, it just gives us the result of the "print it" as if the code had been there all along! Now we've got an activity but we don't know which one! Let's print the title to be sure we got the right one.



It works!

Take Aways

This little tutorial had two or three points. Let's make sure I conveyed them all adequately.

1. Getting familiar with the standard windows.

There are five or six windows you need to be familiar with to use Smalltalk: the Browser, the Workspace and the Debug window are the most important, followed by the Inspector, the Transcript and the Explorer. Each one of these has a specific role:

- the Browser is where the code is browsed, written and organized
- the Workspace is a scratch pad for writing code or notes
- the Debug window appears whenever something goes wrong and lets you inspect the code and change it during execution
- the Inspector shows you the state of an object and lets you send it messages
- the Transcript is for writing debug messages for yourself
- the Explorer is a hierarchical inspector that lets you drill down into the tree of objects you have created

Unlike other systems, in Smalltalk there are many simpler tools that work together rather than one large and complex tool that does everything. Each of these is fairly simple, and you can have as many of each them open as you want or need (although each Transcript will show exactly the same stuff).

Unlike other systems, all of these tools are open for inspection and modification at all times. Almost everything in Smalltalk is written in Smalltalk; you can direct messages to the class `Smalltalk` to get lists of classes and things, which is how the Browser is actually implemented.

2. Understanding the process of programming in Smalltalk

Everybody has a different approach to programming, even within the same language. My approach with Smalltalk is to have a Workspace open as a little scratchpad and do little experiments with it. Whenever I make something that seems useful (like `isExpensive`) I move it into the object I'm working on in the Browser. Whatever changes you make are immediately visible everywhereâ€"remember, you added `isExpensive` and then immediately were able to use it in the Workspace.

Unlike basically every other programming environment (besides spreadsheets), there is no "edit, compile, debug" cycle with Smalltalk; your changes are instantly used and you can edit from directly in the debugger, as we did when writing the `min:` method.

One thing we didn't really touch on much is that a lot of the time with Smalltalk you go hunting for things that already exist in the image. It would be easy to assume that Collections make you walk through them to find the minimum element, for instance. The only ways to find out about `min` are to be lucky enough to read a book that goes into that much detail, or to actually just browse the class. The latter is how you'll find out about 95% of the system. Smalltalk is a simple language but it has a lot of code in the standard image. On the bright side, most of the code in there is pretty simple.

3. Understanding Smalltalk's approach to typing

You've probably noticed that Smalltalk doesn't ask you for the types of your values and methods the way C does. In fact Smalltalk goes further than that and makes assumptions about what will work, leaving it up to the programmer to arrange for things to work. As you saw with `min` (and `value:` if you read the diversion), oftentimes one class will assume that another class can handle a certain message. The disadvantage of this is that the programmer will discover the problem when they get a Debug window. The advantage, though, is that you're able to write code that leverages those assumptions. The collection `min` message assumes that the objects in the collection implement `min:`; if they do, it works, and the collection doesn't know anything about how or why it works, just that it does. Similarly, `collect:` doesn't care if the parameter is actually a block or not; it could be anything that happens to handle `value:` messages. In most cases that will be a block, but it could also be a symbol thanks to that neat hack, or anything else, including your own special classes, as long as they follow the protocol.

This is called "duck typing" in Python, from the expression "if it walks like a duck and talks like a duck, it's a duck," in contrast to statically typed languages like C and Haskell where something must *be* a duck to count as a duck. In Smalltalk, any two things work together if they can talk to each other. They don't have to be soulmates like they do in C.