

Smalltalk B

In the last Smalltalk lab, you wrote code which contained standard language structures (selection, loops, assignment, math, function calls, etc...). During this lab, we will explore the dynamic typing features of Smalltalk.

Getting Started

Start Squeak on the `SmalltalkLab.image` file. Remember that it is okay to keep local backups of your image file but that you should *not* add your resulting image file to SVN.

We'll be looking at various methods in the Browser. I will refer to them by their full class category, class name, protocol, and method name so that you can find them. For example, `Kernel-Numbers >> Integer >> arithmetic >> +` means the `Kernel-Numbers` category, the `Integer` class, the `arithmetic` protocol, and the `+` method.

Important Commands

Alt-. Stop an infinite loop
Ctrl-d do
Ctrl-p print
Ctrl-s submit a method or class
Ctrl-b browser

Remember that using Ctrl-s to submit a class or method is not the same as saving your files. You will need to use the `fileOut` method if you actually want to save your work for submission. For backup images, you will need to use the `SaveAs` link as described in the previous lab.

Dynamic Typing

Dynamic typing is the property of languages where type checking is performed mostly at run time. Types in a dynamically typed object-oriented language are assigned to actual objects rather than to variables that contain references to objects. The type of a variable is inferred at run-time from the type of the object currently assigned to it.

The following workspace commands demonstrate this feature of Smalltalk.

```
x := 3.  
x := 'hi'.  
x
```

Do the following with the above commands:

1. “Do” the first command.
2. “Print” the third command. Notice what prints.
3. Erase the printed output.
4. “Do” the second command.
5. “Print” the third command. Notice what prints. Is it the same as the first print?

Following the above steps, first you assigned the object 3 to `x`. Since 3 has type `Integer`, `x` had type `Integer` when you printed it. Then, you assigned the object `'hi'` to `x`. Since `'hi'` has type `String`, then `x` had type `String` when you printed it the second time. Are you following?

Although any type of object can be assigned to a variable, the only way to access or modify the object is to send it a message. Invalid messages are detected and prevented at runtime. The fact that the invalid messages are caught is what makes this a *dynamically typed* language. Otherwise, you would just proceed to let your program do something as a result of that message send, consequences be damned (or darned for you sensitive folks).

Change your workspace command so that the last one now tries to send the `factorial` message to `x` as in below. Repeat the steps.

```
x := 3.  
x := 'hi'.  
x factorial.
```

The first time you send the `factorial` message, it succeeded, right? An `Integer` knows how to respond to a `factorial` message, which makes sense. However, given that `x` refers to a `String` in the last step, the `factorial` selector winds up being an invalid message the second time. Thus, you get a big ugly error in your window.

Smalltalk is not the only dynamically typed language around. Python, Perl, Javascript, and Lisp are other examples of dynamically typed languages.

What is Smalltalk: strongly typed, weakly typed, or untyped?

In the example we saw above, Smalltalk is able to give `x` an `Integer` type when running the first line and a `String` type when running the second line. We might think this means Smalltalk is untyped, but when we printed `x factorial` the second time, Smalltalk gives us a `MessageNotUnderstand` message saying `factorial` is undefined for `Strings`.

This tells us that Smalltalk is *not* untyped.

That doesn't explain whether it's strongly vs. weakly-typed, though. To figure this out, let's examine mixed-mode arithmetic. Clear your input workspace and input the following statement.

```
3 + 1.7. (Ctrl-p) 4.7
```

Here we can see that Smalltalk added an `Integer` and what appears to be a `Float` and gave us back a `Float`. Must mean its weakly typed, right? This is a coercion, right? Didn't it just do an implicit *type cast* behind the scenes? An implicit type cast would indicate a weakly typed language.

Open your browser to `Kernel-Numbers»Integer»arithmetic»+`. There's actually a shortcut to do this, so don't rush off with your clickety-clicking mouse just yet! Highlight the `+` operator and hit `Ctrl-b`. This should pop up an `Implementors of +` window where you can choose the receiver. Remember, Smalltalk has no idea what the value of `x` will be until it actually evaluates it, so it cannot just assume you meant to go to `Integer's +` operator. Left click on the `Integer` version anyway, and then left-click the `browse` button. Now you didn't have to go through all of the hassle of the `Browser` to get to the method you wanted.

As you can see now, the `+` method is defined for non-integers as

```
aNumber adaptToInteger: self andSend: #+
```

We can fill in our values and we get the following

```
1.7 adaptToInteger: 3 andSend: #+
```

The `#` preceding the `+` character tells us that this is the globally unique identifier meaning `+`, but it should not be evaluated as a message at this time.

You actually used this same technique in the last lab (the `pondChorus` example) when you printed a list of global identifiers by first casting them to `Strings`.

To see what this message does to a `Float`, open the browser to `Kernel-Numbers»Float»converting»adaptToInteger:andSend:.` Rather than converting `1.7` to an `Integer` object as the name suggests, this method converts the `3` to a `Float` object! Perhaps this method might be described better as "work with an `Integer`". This method first performs `rcvr asFloat` where `rcvr` is the `Integer` value. With the `3` object, this returns a `3.0` `Float` object. It then sends the following message to the `3.0` object:

```
3.0 perform: #+ with: 1.7
```

Look in the methods for the Float class. You won't find `perform:with:` listed. Smalltalk is an object oriented language. Click on the hierarchy button to see Float's class ancestors. Browse through the class lineage on the left side for the implementation of the `perform:with:` method.

In effect, this program simply runs by sending the message given as the argument to the selector `perform:` with the argument given as argument to the selector `with:`. Translating, this is equivalent to

```
3.0 + 1.7
```

This is, therefore, *not* an implicit type cast because the code to perform the cast was explicitly found (and could be written or edited by you). This tells us that Smalltalk is *not* weakly typed.

Smalltalk is actually strongly typed. All conversions between types must be explicitly stated in code. Only legitimate messages to a receiver will be answered; all other messages will produce an error.

Problem 1:

Answer the following question in your `handin.txt` file for this lab.

What class implements the `perform:with:` method that the Float receiver was sent?

Fun with Messages: String and Integer concatenation

```
'hi' , ' there'. (Ctrl-p) 'hi there'  
3 , 'hi'. (Ctrl-p) error MessageNotUnderstood ,
```

What's happening? Concatenation in Smalltalk uses the `,` operator (comma). We cannot seem to concatenate a Integer and a String successfully (yet).

We want the concatenation of an Integer and a String to result in a String. We'll need to convert Integers to Strings before we can tackle the larger problem of concatenation. The `asString` method converts instances of Objects to Strings.

```
3 asString (Ctrl-p) '3'
```

Problem 2:

Note: You might crash Smalltalk here if you make a mistake. Just start over with a fresh virtual machine if need be.

Add a method named `,` (yes, the symbol for a comma) to the printing protocol of Integer that takes one argument. When writing an "operator" method, you do not use a colon after the message name (i.e., `, anObj` vs. `,: anObj`). It converts the receiver to a String and concatenates the Strings. (Hint: answer the question "What method converts an Integer to a String?") Test your method as such:

```
3 , ' hi'. (Ctrl-p) '3 hi'  
'hi', ' there'. (Ctrl-p) 'hi there'  
3 , ' hi', ' there'. (Ctrl-p) '3 hi there'
```

When you have successfully implemented your `,` method, you might want to save a backup image.

We've gotten Integer `,` String to do string concatenation between two different types of object. However, this still will not work for other numbers (like Float or Fraction objects). Does the reverse work (String, Integer)? Why/why not? In Java, any Object will be coerced to a String when used with the concatenation operator. Which class would you need to modify in Smalltalk in order to make the `,` operator a general String concatenation operator? Answer these questions and give explanations in your `handin.txt` file.

Polymorphism

Polymorphism is the way that the Smalltalk language allows methods of the same name to have predictable and meaningful results in related instances, yet perform the operations differently to achieve the results.

Smalltalk supports polymorphism by allowing methods defined in classes to be overridden with methods of the same name, but different logic, in a subclass. (<http://www.objs.com/x3h7/smalltalk.htm>)

Problem 3:

Note: Again, you might accidentally kill Smalltalk if you make an error here. You might want to backup your image file if you have not already done so.

Let's solve the previous concatenation issue on more than just Integers. Number is the parent class of Integer and includes all sorts of Numbers (Floats, Fractions, etc..).

Add a method named `,` (again the symbol for comma) to the printing protocol of Number that operates just like the one you added to the Integer class (Do *not* remove the method in the Integer class.)

Add a statement to this new method, before the return line, that prints a short message to the Transcript reporting that the comma method in the Number class has been selected (as opposed to the Integer one).

Then run the following to see that your comma operator now works on Numbers. Look in the Transcript window to see that only the Floats produced a message. The Integers used their own version of the comma operator.

```
3 , ' hi'. (Ctrl-p) '3 hi'  
3.4 , ' hi'. (Ctrl-p) '3.4 hi'  
'hi', ' there'. (Ctrl-p) 'hi there'  
3 , ' hi', ' there'. (Ctrl-p) '3 hi there'  
3.4 , ' hi', ' there'. (Ctrl-p) '3.4 hi there'
```

Your Transcript should show

```
Running the Number , operator  
Running the Number , operator
```

If you see more than 2 responses in your Transcript, then you have errors in your code.

In addition, methods of the same name can be defined in a total different subclass hierarchies. Notice that both the Integer and Float classes have a method named `asFraction` which allows them to be converted to Fraction objects, but that the superclass Number does not.

When you complete this task, you might want to consider backing up your image again. Just in case. (P.S., this is the last warning you're going to get from me on backing up. When you complete a task, just back up already!)

Problem 4a:

Yet another facet of polymorphism in Smalltalk allows a single method to operate on multiple types with the same code.

Add a method named `length` to the comparing protocol of the Integer class. It will take no parameters. It will return the number of digits in the receiver. This method may be iterative or recursive but it may *not* call the `asString` method used previously. (You might want to look up what the arithmetic operators on an Integer actually do.)

Test your method on at least the following tests cases.

```
0 length. (Ctrl-p) 1  
3 length. (Ctrl-p) 1  
-3 length. (Ctrl-p) 1  
1234 length. (Ctrl-p) 4  
10300 length. (Ctrl-p) 5
```

Be certain it works on both positive and negative numbers.

Problem 4b:

Add a method named `length` to the `String` class (`Collections-Strings>>String>>accessing>>length`). It will take no parameters. It will return the number of characters in the `String`. Use the `size` method available to `Strings`.

Test your method on at least the following test cases.

```
' ' length. (Ctrl-p) 0
'a' length. (Ctrl-p) 1
'ab c' length. (Ctrl-p) 4
'w3c,t' length. (Ctrl-p) 5
```

Problem 4c:

Answer the following questions in your `handin.txt` file.

What class implements the `size` method that the `String` receiver was sent? Think about how a `String` might be stored in memory. What do the parent and grandparent classes of `String` suggest about the implementation of a `String` in `Smalltalk`?

What would you expect to see as the difference between the answer to these two commands in the workspace? Why?

```
-3 length.(Ctrl-p)
-3 asString length.(Ctrl-p)
```

Problem 4d:

In class, we briefly discussed the notion of late vs. static binding with regard to the values of `self` and `super`, respectively. When you use the value of `self` in a method, it will insert a reference to the object that is currently running the method in its place. Thus, `self` is only “bound” to a value when the method is actually called. However, `super` is bound to the superclass of the defining class at the time a method is submitted. No matter what object is executing the method, `super`’s value will always be the superclass of the class in which the method code is defined.

Rewrite the `length` method for `String` so that it uses the `super` construct.

What implications does this have for subclasses of the `String` class? Give a full explanation in your `handin.txt` file.

Problem 4e:

Create a `LabB` class category and a `LabB` class. Create a method protocol named `problem` in your `LabB` class.

Add a method `LabB>>LabB>>problem>>length` that takes an argument and returns its length using the `length` methods you just wrote. Do not test for the type of the argument.

Test your method on at least the following test cases.

```
x := LabB new. (Ctrl-d)
x length: 0. (Ctrl-p) 1
x length: 3. (Ctrl-p) 1
x length: -3. (Ctrl-p) 1
x length: 1234. (Ctrl-p) 4
x length: 10300. (Ctrl-p) 5
x length: ' '. (Ctrl-p) 0
x length: 'a'. (Ctrl-p) 1
x length: 'ab c'. (Ctrl-p) 4
x length: 'w3c,t'. (Ctrl-p) 5
```

Submitting your lab

Submit the following files to SVN for grading:

- `String.st`
- `Number.st`
- `Integer.st`
- `LabB.st`
- `handin.txt`

Verify that your various `.st` files load correctly *and* contain your changes before declaring that you are done with lab.