CS 335 - Homework 8

Deadline:

Due by 11:59 pm on Friday, April 22nd

How to submit:

Submit your files for this homework using `~st10/335submit` on nrs-labs, with a homework number of `8`

Purpose:

To think about several classic parameter passing methods, and to start getting used to working with Squeak.

Important notes:

- You are expected to use the Squeak implementation of Smalltalk (available from www.squeak.org).

The Problems:

Type your answers for Problems 1-4 into a file named `hw8-params.txt`, and submit that file.

Problem 1:

(adapted from MacLennan Exercise 2-15, p. 60)

Consider the following FORTRAN subroutine, whose name is `TEST`, and which has three parameters, `X`, `Y`, and `Z`:

```
SUBROUTINE TEST (X, Y, Z)
X = 2
Z = X + Y
RETURN
END
```

And, consider the following code fragment, invoking the `TEST` subroutine with the arguments `N`, `N`, and `M`:

```
N = 5
CALL TEST (N, N, M)
```

1 part a

What will be the final values of `N` and `M` after this invocation of `TEST` if the parameters are **passed by reference**?

1 part b

What will be the final values of `N` and `M` after this invocation of `TEST` if the parameters are **passed by value-result**?

Problem 2

Remember that, unless specified otherwise, parameters in Algol-60 are **passed by name**.

Consider this example Algol-60 program, adapted from:

http://courses.cs.vt.edu/~cs3304/Spring00/notes/Chapter-8/tsld018.htm

```
begin
    integer i, j;
    integer array A[1:3];

    procedure sub1(x, y);
        integer x, y;
        begin
            x := 1;
            y := 2;
            x := 2;
            y := 5;
        end;

    i := 1;
    A[1] := 7;
    A[2] := 14;
    A[3] := 21;

    sub1(i, A[i]);

    Print(i);
    for j := step 1 until 3 do
        Print(A[j])
end
```

2 part a

Assuming that `Print` prints the value of its argument on its own line, write out what this program will print to the screen when executed.

2 part b

But -- what if Algol had passed its parameters using pass-by-reference, instead of pass-by-name? What, then, would this program print to the screen when executed?

Problem 3

Now consider this Algol-60 program, adapted from:

http://courses.cs.vt.edu/~cs3304/Spring00/notes/Chapter-8/tsld019.htm

```
begin
    integer i, j, k;

    procedure sub1(x, y, z);
        integer x, y, z;
        begin
            k := 3;
            y := x;
            Print (y);
            k := 7;
            z := x;
            Print (z);
        end;

    sub1(k+1, j, i);
    Print('looky');
    Print(i);
    Print(j);
    Print(k);
end
```

Write out what this program will print to the screen when executed.

Problem 4

(adapted from MacLennan, Exercise 10, p. 141)

Consider the following Algol-60 program.

```
    begin
        integer i, j;
        integer array A[1:3];

        procedure P(x, y);
            integer x, y;

            begin
                y := 2;
                Print(x);
                i := 3;
                Print(x);
                Print(y);
            end;
```

```
        A[1] := 4;
        A[2] := 8;
        A[3] := 12;
        i := 1;

        P(A[i], i);
        P(i, A[i]);

        for j := 1 step 1 until 3 do
            Print (A[j])
end
```

Assume that the Print in this Algol's library prints its requested value on its own line. What values are printed by this program if you assume that:

4 part a

...`x` is passed by value and `y` is passed by value.

That is, assume that the function begins with:

```
procedure P(x, y);
    value x, y;
    integer x, y;
```

What is printed by this program?

4 part b

...`x` is passed by value and `y` is passed by name?

That is, assume that the function begins with:

```
procedure P(x, y);
    value x;
    integer x, y;
```

What is printed by this program?

4 part c

...`x` is passed by name and `y` is passed by value?

That is, assume that the function begins with:

```
procedure P(x, y);
    value y;
    integer x, y;
```

What is printed by this program?

4 part d

...`x` is passed by name and `y` is passed by name?

That is, assume that the function begins as originally shown, with:

```
procedure P(x, y);
    integer x, y;
```

What is printed by this program?

Problem 5:

Getting started in Squeak...

* Squeak should be installed in the BSS 313 lab, and *may* be available in BSS 317, LIBR 121, and LIBR 122. You can also download it from www.squeak.org.

  Be sure, as you work in Squeak, to save your **.image** and **.changes** files somewhere safe - for example, NOT on a lab machine that clears itself between logins...! - so that you can restore your work if you need to re-install Squeak later, or work on a different computer, etc.

* oy -- how should I describe WHICH click I mean, here? Recall the "chart" from lecture:

Color	Mac mappings	Windows mappings	usual "meaning"
Red (left)	click button	left-click	move/select
Yellow (m)	option-click	right-click	context menu
Blue (rt)	command-click	ALT-left-click	window/Morphic

  In this handout, I'll try using Color/Windows/Mac to indicate a which click I mean -- red/left-click/click, yellow/right-click/option-click, or blue/ALT-left-click/command-click. Please let me know if I should *not* do this for the Homework 9 handout... 8-)

* Start up Squeak (often double-clicking on a .image file, such as **Squeak3.10.2-7179-basic.image**, seems to work), and within the Squeak VM window that appears, red/left-click/click and select the menu item **save as...** to save an image under a DIFFERENT name of your choice.

* Note that you can red/left-click/click, select the menu item **open...**, and select the submenu item **workspace** to open up a Workspace object, or if you prefer you can drag a Workspace object from the Tools tab on the the right-hand side of the Squeak VM window.

  Likewise, you can red/left-click/click, select the menu item **open...**, and select the submenu item **transcript** to open up a Transcript object, or if you prefer you can drag a Transcript object form the Tools tab on the right-hand side of the Squeak VM window.

  And, while we'll discuss this more next week, if you would like to browse through all of the classes and objects that already exist within Squeak, use either of these methods to obtain a System

Browser object, EXCEPT note that the System Browser is called a "class browser" in the **open...** submenu, and indeed is the first item in that menu, and the System Browser is called simply "Browser" in the icons in the Tools tab, and it was the last icon in the 2nd column in my Tools tab. (Why the different names for the same thing? Good question...!)

* Within a Workspace object, recall that you can:
  * highlight an expression OR red/left-click/click to the right of an expression in the Workspace,
  * then yellow/right-click/option-click to bring up a menu,
  * and select **print it** to send a print it message to print the value of that expression.

  * (or, on my Mac,
    highlighting an expression or red/left-click/clicking to the right of an expression
    and then typing ctrl-p also sent that expression a print it message, as was so helpfully pointed out in lecture...)

Likewise, if you simply want to **perform/"do"** that expression, you can highlight or red/left-click/click to the right of an expression in the Workspace and yellow/right-click/option-click it to bring up a menu and select **do it** to send a do it message to perform that expression. (or do ctrl-d at that point...) (This is especially useful for an expression for which you are more interested in, say, its side-effect(s) than its value...)

ADDITIONAL RESOURCE NOTE:

I have found an interesting Squeak tutorial, "Writing Simple Games in Squeak", that introduces you to Squeak's Morphics graphics system (and really to the Squeak environment) by walking you through the development of a game:

http://squeak.preeminent.org/tut2007/html/

I'm not assigning this as required reading, yet, because I haven't gotten through the whole tutorial myself yet. But it looks promising, and includes screen dumps that seem very close to what I get as I walk through the tutorial myself. It also stresses test-driven development, which is a REALLY good idea, anyway.

I recommend that you check it out.

5 part a

Recall how, in lecture, we discussed that a Smalltalk-80 statement such as:

```
39 gcd: 26
```

...is sending a `gcd:` message with object `26` as its argument to the object `39`.

We also mentioned that a string literal object in Smalltalk-80 is surrounded by **single** quotes.

Open up a Workspace object and a Transcript object within Squeak.

In that Workspace object, use the `show:` method of the Transcript object to send a `show:`

message to the Transcript object with a greeting of your design that also includes your name. (hint: you are more interested in doing this than in seeing its value...)

If this has gone well, you should see your greeting now in the Transcript window.

5 part b

Being careful NOT to overwrite your expression from part a, now experiment with additional expressions in the same Workspace window.

Send `show:` messages to the Transcript object with **at least three additional expressions** (make sure that at least two are NOT string literals!). These could be expressions from lecture, or from Smalltalk-80 reading that you do, or from sheer experimenting -- your choice. The more you experiment, the better.

5 part c

One can save the current contents of a Workspace or Transcript to a text file.

Within one of these, yellow/right-click/option-click to bring up a menu. Select **more...** because the command we want isn't showing yet. Look for **save contents to file...** near the bottom of the next menu that appears. When selected, you can now specify a file name to which to save that Workspace's or Transcript's contents.

At this point, save the contents of your current Workspace and Transcript, naming the workspace contents you've saved as `hw8w-5abc.txt` and the transcript contents you've saved as `hw8t-5abc.txt`. These are the two files that you will submit for parts a, b, and c.

5 part d

Note that these `.image` and `.changes` files are rather large. How large are they?

First, quit Squeak (red/left-click/click, and select **save and quit**). Then, find you saved **.image** file, and in that same directory should be a corresponding **.changes** file, too.

Look at how big they are, and type the size of each (file name, and then its size, for both your .image and your .changes file) into a file `hw8-5d.txt`; submit this file.

5 part e

Note that the **class** unary message, when sent to an object, returns the **name** of the class of the receiving object.

Open up a new Workspace object, and send **class** messages to each of the following objects to find out the class of each. Use **print it** for each such message to show the value of that expression to the right of that expression, being careful not to delete that result when you go to the next line for the next expression. ASK ME if you are not sure what I mean here, or if you have trouble with this!

Note: be careful with the results of precedence in Smalltalk-80. Make sure that you are sending the **class** message to the desired receiving object for each of your expressions, using parentheses as

needed.

Send a **class** message to...

* ...an object that is your name expressed as a Smalltalk-80 string literal

* ...an object that is a # followed by a sequence of letters of your choice (and no blanks)

* ...the object `13`

* ...the object `3.7`

* ...the object `true`

* ...the object `True`

* ...the object `5@8`

* ...the object `5<8`

* ...the object `#('a' 1 #george)`

* ...the object `#('a' 1 #george) at: 1`

* ...the object `#('a' 1 #george) at: 3`

* ...the object `$x`

* ...the object `#croak`

At this point, save the contents of your current Workspace, naming the workspace contents you've saved as `hw8w-5e.txt`, and submitting this file.

5 part f

Another potentially-useful object is the **Method Finder**. It is available via the same means as transcripts, workspaces, and system browsers, and it can be used to, well, find out about what methods are available, or what methods might lead to a desired result.

For example, in the top left pane of a Method Finder object, if you type part of a method name and then type return, then in the pane below will be listed method names that include that fragment. Try it with **copy** -- there are LOTS of method names that contain **copy** or **Copy** within their names! (Notice, too, how **n-ary** method names are written --- **copy:from:to:rule:**, for example, is a 5-ary method. And unary method names are written without any colon --- **class**, for example, for that unary method.)

Want to know what class(es) of objects can accept such messages? Click on the method in the pane underneath the top-left pane, and the top-right pane will show what class of object accepts such a message. If you click on **copy:from:to:rule:**, for example, you can see that objects of the **Form**

class can accept **copy:from:to:rule:** messages.

As one final example, in the top left pane of a Method Finder object, if you type an object, then a period and a blank, then an argument, then a period and a blank, then the next argument, then a period and a blank, ... then the result, and finally type return, then the methods that can lead to that result with that receiving object and arguments will appear in the pane below.

(If you are looking for a unary method, then typing the receiving object, then a period and a blank, and then the result, followed by typing return, will suffice.)

For example, if you type

```
3 . 5 . 8
```

...in that top-left pane, followed by return, then the pane beneath shows that method + can be used to obtain that result with those objects.

And, if you type

```
39 . 26 . 13
```

...in that top-left pane, followed by return, then the pane beneath shows that methods **-** , \\ , **gcd:** , and **rem:** can be used to obtain that result with those objects.

As some practice using this (and to explore some of the many classes and methods included in Squeak):

* Think of at least 3 strings that you would like to know if there are methods whose names include (besides **copy**! 8-)). Try them out in a **Method Finder** object.
* In a transcript object,
  * paste/type in your **three** favorite such strings,
  * and after **each**, type the names of at least 3 method names including that string;

  * and, for each of those methods you chose, type to the right of that method name (in your transcript object) the name of **at least one class** that can accept such messages.

(That is, for **copy**, I could type into such a transcript object:

```
copy
copy:from:in:    BitBlt
copy:addArg:     MethodFinder
addCopyItemsTo: Morph
```

...and please put a blank line after this, for easier readability.) (So, you had better have at least 3 "sections" of 4 lines each after this part.)

Ask me if you are not sure what I am asking for here!

Now use a method finder to find **at least one method** that can be used to perform each of the following. For each of the following:

* paste what you used in the Method Finder's top left pane into a Transcript object,

* and on the next line of that Transcript object type in the name of at least one method that you discover using the Method Finder.
* (Please type in a blank line in that Transcript object between each such pair of lines.)

For example, for the examples described earlier, you'd paste/type at least the following in your Transcript object for these parts:

```
3 . 5 . 8
+

39 . 26 . 13
rem:
```

Use a Method Finder object to find at least one method:

* ... that can be used to send a message to a string such as **'squeak'** to result in **'kaeuqs'**

* Can be used to send a message to the class `Color` with arguments 1, 0, and 0 to result in the object `Color red`

* Can be used to send a message to a string such as **'squeak'** to result in **'Squeak'**

* Can be used to discover that 8 to the 3rd power is 512

* Can be sent to #(1 2 3 4) with argument [:x| x odd] to result in #(1 3)

* Can be sent to #(1 2 3 4) with argument [:x| x odd] to result in #(2 4)

* Can be sent to #(1 2 3 4) with argument [:x| x odd] to result in #(true false true false)

* Can be sent to **'squeak'** with argument 3 to result in **'squ'**

* Can be sent to **'squ'** with argument **'eak'** to result in **'squeak'**

* Think of at least **three** more "patterns" you are interested in knowing if there are methods for; put your three favorite such patterns, and at least one method that results in them, in your transcript object.

At this point, save the contents of the resulting transcript object, naming the workspace contents you've saved as `hw8t-5f.txt` and submitting this file.

5 part g

At this point, you should have been exposed to at least some additional methods. Open up a workspace, and type in at least 10 Squeak expressions/messages:

* include at least 10 different methods not covered in lecture nor specifically mentioned in this homework handout (although ones you learn about doing the previous problems in this homework are fair game!);

* make sure at least one of these is a unary method; make sure at least one of these is an n-ary method for which n is 3 or greater;

* use **print it** for each such message to show the value of that expression to the right of that expression, being careful not to delete that result when you go to the next line for the next expression.

* (if you find a message for which you believe it is "shown off" better by using **do it** , then **also** do that; if any of these result in something going to a transcript, submit the contents of that transcript object, also, naming it `hw8t-5g.txt`. If one "does" something that isn't visible, also describe what it did in that transcript object.)

Save the contents of the resulting workspace object, naming the workspace contents you've saved as `hw8w-5g.txt` and submitting this file. (Only submit the transcript contents as `hw8t-5g.txt` if it is applicable for your messages/expressions.)

Nifty methods you find may be collected and passed out to all...