CS 335 - Homework 9

Deadline:

Due by 11:59 pm on Friday, April 29th

How to submit:

Submit your files for this homework using `~st10/335submit` on nrs-labs, with a homework number of `9`

Purpose:

To get practice creating classes and methods in the Squeak implementation of Smalltalk-80.

By the end of doing this homework you should:

- Understand the functions of the most common windows in Squeak
- Be able to create classes in the System Browser
- Be able to create method categories and write code for methods in the System Browser
- Be able to use the Workspace to test your code

This was adapted from:

http://www.cosc.canterbury.ac.nz/wolfgang.kreutzer/cosc205/smalltalk1.html

http://www.cc.gatech.edu/classes/AY2000/cs2803ab_fall/labs/IntroLab.html

http://www.cc.gatech.edu/classes/AY2000/cs2803ab_fall/labs/MuppetLab.html

Important notes:

- You are expected to use the Squeak implementation of Smalltalk (available from www.squeak.org).
- To describe which variety of 3-button mouse click is expected: 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
```

  This is the chart I gave in class, and that is in my notes; HOWEVER, I am finding that whenever a yellow button click is asked for, I'm having to use what I thought was called command-click on my Mac to get it to happen (clicking while pressing the apple-key); if you are using a Mac, please give this a try if you are having trouble in this situation.

  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.

- We're seeing that Squeak tends to provide multiple ways to do things; here I am giving one way, but I suspect there are other ways of doing many of these actions as well.

The Problems:

Problem 1:

The purpose here is simply to walk through the process of defining one's own classes.

Domain: "a small segment of Sesame Street, a well known urban neighborhood populated by a wide range of rather exotic creatures"

...but we'll start simply, with just two classes: `Monster` and `CookieMonster`

- (note that Smalltalk class names are expected (convention) to be written in CamelCase starting with an upperclass letter -- as in Java...)

`Monster` will be an abstract class, describing "a number of aspects common to monsters of all walks of life", and `CookieMonster` will be a subclass of `Monster`.

In modeling monsters for our purposes, what do we care about? We'll start simply: monsters are characterized (to start) as a `colour` and a `tummy`. The `colour` will be stored as a symbol, and at creation monsters are equipped with an empty `tummy` instantiated to a predefined data structure called `Bag`.

- By the way: in C++, you probably called `colour` and `tummy` the **data fields** for class `Monster` -- the Smalltalk references being used for this homework call them "**instance variables**".

Monsters should be able to eat, and to answer whether their tummy is empty. These, then, will be methods of the `Monster` class.

- You should be used to the concepts of accessor methods (to obtain the value of data fields) and modifier methods (to modify the value of a data field) from C++ object-oriented programming; in Smalltalk, the **instance methods**, or methods for an instance of the class, are categorized as **initialization**, **access**, **queries**, and **actions**.

By the way -- since Smalltalk classes are also objects, they may also have **class methods**, or methods for the class itself, in addition to instance methods. For example, a class should have a class method for creating a new instance of the class -- so `Monster` will need this.

What is special about CookieMonsters in particular? Well, they continually nag for cookies, which are the only food they will eat. Once awakened they may only be silenced once their immediate hunger has been stilled, after which they will fall asleep until some foolish user wakes them again. So, `CookieMonster` has two additional data fields/instance variables, `state` and `hunger`. Its `state` is a symbol, either `#awake` or `#asleep`, and its `hunger` is a number, determined at random when it wakes up.

`CookieMonster` should override `Monster`'s `eat` instance method, since they only like to each cookies. As they are not particularly strong, they must also know how to solicit food by begging, and the `nag` instance method will offer this functionality. And, there will be instance methods for testing items prior to digestion, and for describing how a `CookieMonster` instance will beg.

Now: to implement all this in Squeak, we will:

- first make a new **class category**, `SesameStreet`
- add the `Monster` and `CookieMonster` classes to the `SesameStreet` class category
- define and test (or at least exercise) all of these class' methods

We'll use a `Browser` to do the above, and use a `Workspace` along with another type of view, called `Inspector`, for testing/exercising purposes.

Creating a new class category

- open up a System Browser (called just a Browser in the Tool menu)
- to create a new class category: yellow/right-click/option-click in top-left browser window, select "add item" option; (fill in desired name, here `SesameStreet`, in the dialog window that should appear) and accept;
- Classes can be defined once the new class category has been added to the system.

Adding a new class to a class category

- In the top-left Browser window, select (red/left-click/click) the desired class category that is to have a new class
- Now the bottom Browser pane will show a template for a class definition, which you can fill in as desired:

```
Object subclass: #NameOfSubclass
     instanceVariableNames: ''
     classVariableNames: ''
     poolDictionaries: ''
     category: 'SesameStreet'
```

  - Notice how the class category is already filled in for you.
  - You don't have to fill everything in -- for example, we won't be filling in anything for poolDictionaries at this point.
  - Like Java, notice that there is an `Object` superclass that a class is a subclass of if it isn't otherwise explicitly a subclass of another class.
  - We'll edit this class definition as follows:

```
Object subclass: #Monster
     instanceVariableNames: 'colour tummy'
     classVariableNames: ''
     poolDictionaries: ''
     category: 'SesameStreet'
```

  - enter a comment describing this class in the lower portion of this pane:

```
"This ABSTRACT class implements some generic structure and behavior
common to different types of monsters in the world of Sesame Street."
```

  - I then command-clicked (but, I think this should be yellow/right-click/option-click) to get a menu with an **accept** option; select this to save your new class.
  - Choosing **accept** from the menu compiles the definition and adds it to the dictionary of classes the system knows about.
  - Hopefully, you now see `Monster` in the top-row-second-windowpane of the System Browser, here indicating that it is a class in class category `SesameStreet`.
  - Note that choosing "update" from the leftmost browser pane's yellow/right-click/option-click button menu may sometimes be needed to force a browser to show the most recent changes.
- Now set up a class for `CookieMonster`
  - In the top-left Browser window, select (red/left-click/click) the desired class category `SesameStreet`. (To "get out" of the Monster class, I clicked on Monster's class name in the second pane, and then re-clicked on SesameStreet in the first pane. Then there was a new class template in the lower window to fill in.)
  - When filling in this class definition, we want to specify that `CookieMonster` is a subclass of `Monster`:
- Now the bottom Browser pane will show a template for a class definition, which you can fill in as desired:

```
Monster subclass: #CookieMonster
     instanceVariableNames: 'state hunger'
     classVariableNames: ''
     poolDictionaries: ''
     category: 'SesameStreet'
```

  - enter a comment describing this class in the lower portion of this pane:

```
"This implements a monster who only eats cookies and is typically
very hungry"
```

  - command-click (but, I think this should be yellow/right-click/option-click) to get a menu with an **accept** option; select this to save this new class.
  - Hopefully, you now see `CookieMonster` in the top-row-second-windowpane of the System Browser along with `Monster`

Creating a class method

- Typically, we need to create access methods for a class first. Let's do that for Monster.
- Select Monster in the second browser pane
- In the third browser pane, I command-clicked (but, I think this should be yellow/right-click/option-click) and then select "new category".
  - select "new...", and create a new category `access`
  - (select this new category in the 3rd pane, if it isn't selected already. You should see a method template in the bottom pane, now)

- We'd like a `colour` method that returns the value of the `colour` instance variable/data field:

```
colour
   "return this monster's colour"
   ^colour
```

- and yellow/right-click/option-click(command-click?) and accept. Enter your initials if requested.
- Hopefully, colour now appears in the fourth pane in the top row.

Adding more to the Monster class

- We also want an access method for the tummy; to add another method to the same method category, I usually have to type on something else -- say Monster -- and then on access, and now enter a tummy accessor method:

```
tummy
    "return this monster's tummy"
    ^tummy
```

- don't forget to **accept** the newly-entered method!

- Now add methods to method category `access` to change/set the `colour` and `tummy` instance variables/data fields. Since the user presumably should be able to indicate what these should be set to, these should be keyword methods of arity 2:

```
colour: aSymbol
    "set the monster's colour"
    colour := aSymbol.
    ^self
```

- Note: the parameter name `aSymbol` here is actually a significant style example; the Squeak Swiki (!) notes that:

> "Since Smalltalk is a "dynamically typed" language one should take care to document clearly what kinds of objects a method expects as its arguments.
>
> Good Smalltalk style uses typenames prefixed by "a" or "some" (e.g. "anInteger", "someMonster") for this purpose, instead of less descriptive identifiers such as "x", "y" or "fred".
>
> This convention for naming arguments uses the name of the most general class of those objects a method is willing to accept. If we don't want to make any assumptions we can use : "someObject" or "someItem"."

- This is a CS 335 course style standard for Squeak, then.
- And now let's add a way to set a tummy.

```
tummy: aCollection
    "set a monster's tummy"
    tummy := aCollection.
```

```
    ^self
```

- In Smalltalk, you need those means to access and set your data fields before you can really create ways to create instances of your class...!
- But, we have done so -- so we want an **initialization** method category (select Monster class, yellow/right-click/option-click(command-click?) on the 3rd pane, select "new category", select "new...", type in initialization, and accept)
  - In this initialization method category, make a method initialize:

```
initialize
    "create a new monster"
    self colour: #green.
    self tummy: Bag new.
    ^self
```

Creating a class method

- We would like one class method category. To add it, you need to select Monster in the 2nd pane, and then click the little class button beneath the 2nd pane. Now when you yellow/right-click/option-click(command-click?) on the 3rd pane and select "new category", you are adding a new class method category; select **instance creation** for the category, and give it the following method named **new** as so:

```
new
   "create a new monster"
   ^super new initialize
```

- (Note that, if you want to view or modify your instance methods for Monster, you can click on the instance button under the 2nd pane while Monster is selected to do so.)
- And now you can TEST your new Monster class:
  - Open a new Workspace
  - Type in that workspace:

```
george := Monster new.

george inspect.
```

  - Highlight these, and `doIt` -- a Monster inspection window should open, and you can inspect your Monster instance!

More methods for Monster

- Continuing in this vein. below are additional method categories and methods for Monster. You should add these to your Monster class.

```
    actions    "instance method category"

    eat: someItem.
        self tummy add: someItem.
        ^self
```

queries "instance method category"

```
isEmpty
   ^ self tummy isNil
```

Adding CookieMonster methods

- Now add these method categories and methods to the CookieMonster class.
  - (Note: the idea here is to think about each line as you type it, so as to get more comfortable with Squeak methods.)

private "instance method category"

```
askForCookie
   ^ FillInTheBlank request: 'Give me cookie !!! (please)'

complainAbout: anItem
    Transcript show: 'No want ', anItem printString.
    Transcript cr.
    self colour: #red.
    ^self

isCookie: anItem
" | serves as the OR operator"
    ^ ((anItem = 'cookie') | (anItem = #cookie))
```

actions "instance method category"

```
eat: aCookie    "overloaded eat:!"
    super eat: aCookie.
    self colour: #green
    ^self

nag
   | item |
      [self isAwake]
      whileTrue:
        [item := self askForCookie.
        (self isCookie: item)
              ifTrue: [self eat: item]
              ifFalse: [self complainAbout: item].
        (self isFull) ifTrue: [self sleep]]
   ^self

sleep
   self state: #asleep.
   self hunger: 0.
   ^self

wakeUp
   self tummy: Bag new.
self state: #awake.
self hunger: (Random new next * 13).
```

```
"Cookie Monsters are superstitious and never eat more than
13 cookies in one go !"
self nag
^self  "?"

queries      "instance method category"

isAsleep
   ^ state = #asleep

isAwake
   ^ self isAsleep not

isFull
   self isEmpty
   ifFalse: [^ self tummy size >= self hunger]    ifTrue: [^false]

access        "instance method category"

hunger
   ^ hunger

hunger: anIntegerNumberOfCookies
    hunger := anIntegerNumberOfCookies.
    ^self "?"

state
   ^ state

state: aSymbol
    state := aSymbol.
    ^self

initialization    "instance method category"

initialize
   self state: #asleep.
   self hunger: nil.
   super initialize
   ^self
```

"To ensure proper initialization Monster's creation class method is also overridden."

```
creation       "CLASS method category"

new
   ^ super new initialize
```

* "You will have noted that most messages are rather short.

In fact, a large proportion of them consists of a single line of code, returning (^) or assigning (:=) some value.

This is typical for object-oriented programs and Smalltalk code in particular, since all valid patterns of access to variables must be explicitely defined.

In the interest of reliability many state variables should not be accessible at all from outside of an object.

Smalltalk's approach of requiring explicit method definitions for any access to variables is facilitated by its programming environment.

Since the browser allows rapid definition of such selectors with only a few mouse clicks, modifying an already existing method, this is not particularly bothersome to do - and it pays in terms of program reliability.

In a traditional listing such methods tend to clutter clutter the code, but browsers reduce the need for such tedious documentation."

* "Note that choosing "fileOut" fom the yellow button menu attached to the class category pane of a browser saves all such class definitions in the selected category as a text file (in the current directory), which can be "read back" into Squeak (i.e. each definition is recompiled) from a file list."

  – (According to the Georgia Tech documentation: Exporting a class definition from Squeak is known as performing a fileOut. Similarly, importing a class definition into Squeak is known as performing a fileIn.)

- SO: at this point, do a fileOut of your SesameStreet class category. The resulting file should be named `SesameStreet.st`
  – So you will know: here is how you can open up such a saved class category:
    * if you were to open up a brand-new Squeak image, and wanted to add SesameStreet to it:
      * red-click (left-click) on the desktop and choose **open...**, and then **file list...**
      * you will then be shown a browser window containing the contents of your directory.
      * select your newly-copied-over **SesameStreet.st** file and yellow-click (right-click) within the file list browser, selecting **fileIn entire file**.

        Squeak will then import the class definition into its class library, and when you look in a System Browser, you should see:
        * class category **SesameStreet** at the bottom of the list in the leftmost-top pane,
        * if you click on class category **SesameStreet**, you should see classes **CookieMonster** and **Monster** in the second pane,
        * if you click on class **Monster** in the second pane, you should see method categories **access**, **initialization**, **creation**, and **actions** in the third pane,
        * if you click on method category **access** in the third pane, you should see methods **colour**, **colour:**, **tummy**, and **tummy:** in the fourth pane, and
        * if you click on method **colour:** in the fourth pane, you should see the body of method **colour:** in the bottom pane.

Problem 2

Now, you will add to the Monster class.

* Add two more class variables to the **Monster** class: **greeting** and **name**.
    * Click on the **Monster** class in the second System Browser pane. Its code should appear in the bottom pane.
    * Add instanceVariableNames of **greeting** and **name** (to go along with **colour** and **tummy**).
    * red-click and select **accept** to accept these new instance variable names.
* Now we need methods for using and setting this **greeting** and **name**.
    * When you click on class **Monster** in the second pane in a System Browser, you see its method categories in the third pane.
    * Let's add a **name** method to the **access** method category. Select **access** in the third pane of the System Browser, and the bottom pane then has within it a method template. For simplicity, simply delete all that and enter the following in that bottom pane:

      ```
      name
          ^name
      ```

      red-click and **accept** to accept this change (**name** should appear in the 4th pane as another **access** method).
    * And, add an analogous **greeting** method to the **access** method category (that simply returns the **greeting** for this Monster instance).
    * And, allow a Monster to change its name - add a **name:** method (also in the **access** method category) with the following body:

      ```
      name: aSymbol
         name := aSymbol.
         ^self
      ```

    * And, allow a Monster to change its greeting - add a **greeting:** method (also in the **access** method category) with an analogous body, except call its method parameter **aString** instead of **aSymbol** (because that will be more convenient for a greeting, which will be more likely to be a phrase).
* Finally, the greeting and name should be able to be initialized.
    * Click on the **Monster** class's method category of **initialization**, and to the existing **initialize** method add statements using the new **greeting:** method to set **greeting** to a default value of 'Hi there!' and using the new **name:** method to set **name** to a default value of **''** (note that's two empty quotes). Remember to **separate** statements within a method using periods, and remember to **accept** your changes (yellow-click in the bottom pane, and select **accept**).
* Consider: you can write to a Transcript by sending it a **show:** message whose argument is, well, the

value of what you want to show. And, you can send a newline/carriage return by sending a Transcript a **cr** message.

Create a **greet** message for class **Monster**, under method category **actions**, that simply writes the calling monster's greeting, followed by a newline/carriage return, to the Transcript.

* Exercise these new additions/modifications. Open up a Workspace and a Transcript. In the Workspace, perform each of the following, yellow-clicking and selecting **do it** after **each** of the first four and yellow-clicking and selecting **print it** after each of the second three (being careful to leave the printed-out result to the right of each of those three) and selecting **do it** after the last one, to cause something appropriate to appear in the Transcript:

  * create a new Monster: **harry := Monster new.**
  * change its name to **#Harry**: **harry name: #Harry.**
  * change its greeting to **'Hellooo!'**: **harry greeting: 'Hellooo!'.**
  * change its colour to **#orange**: **harry colour: #orange.**

  * see its name: **harry name.**
  * see its greeting: **harry greeting.**
  * see its colour: **harry colour.**

  * cause a greeting to be written the the Transcript: **harry greet.**

  Now, for good measure, in the Workspace do:

  harry inspect.

  * ...and yellow-click and select **do it** to inspect your current **harry** instance in that way.

  * Add any additional testing that you would like to the above.

* To show that you've tried at least the steps specified above in the Workspace and Transcript, yellow-click in the Workspace, select **more...**, select **save contents to file...**, type in the name **ws2.txt** and **accept**.

  And, in the Transcript, yellow-click in the Transcript, select **more...**, select **save contents to file...**, type in the name **ts2.txt** and **accept**.

  You should now have text file **ws2.txt** and **ts2.txt** in your working directory; submit these files as part of Problem 2. (Again, you'll be saving a copy of your actual code later on...)

Problem 3

Above, you created Monster and CookieMonster methods, but you only tested the Monster methods. Now test the CookieMonster methods for credit as follows:

* Open up a new Workspace and a new Transcript.

* In the Workspace... (use **do it** for each of these unless specified otherwise)
  * assign a variable **ck** to a new instance of a CookieMonster.

- Make **ck's** name **#Cookie**, and make **ck's** greeting 'HI! ME LIKE COOKIES'
- Send a **greet** message to **ck** (so that you'll see a greeting on the Transcript).
- Send an **isAsleep** message to **ck**, and use **print it** to show its value next to this message in the Workspace;
- Send an **isAwake** message to **ck**, and use **print it** to show its value next to this message in the Workspace;
- Send a **wakeUp** message to **ck**. As it nags for cookies, at least one time enter something BESIDES a cookie (so that you'll see a complaint show up on the Transcript).

  (IF you'd like - after you enter a non-cookie, send a **colour** message to **ck** BEFORE you enter a cookie, or send an **inspect** message to **ck**, and check out **ck's** colour at this point... surprised?)

  You may have to enter a cookie up to 13 times --- if you'd like to reduce the maximum number of cookies it begs for at a time to a number less than that, you may modify the appropriate CookieMonster's method accordingly... 8-)
- Add any additional testing/demonstrations that you'd like.

* To show that you've tried at least the steps specified above in the Workspace and Transcript, yellow-click in the Workspace, select **more...**, select **save contents to file...**, type in the name **ws3.txt** and **accept**. Do the same thing for the Transcript, saving it as **ts3.txt** and **accept**.

  You should now have text files **ws3.txt** and **ts3.txt** in your working directory; submit this file as part of problem 3. (The code will be saved after the last problem.)

Problem 4

IF you were going to create a new class category, you could open a System Browser object, yellow-click over the leftmost-top pane, select **add item...**, type in the desired new class category name, and select **accept** to accept it. But you **don't** need to do that here, because we are going to add classes to **SesameStreet** instead.

To start creating a new class, you need to select the class category it will be within. So, select the class category **SesameStreet** from the leftmost-top pane; you've done it correctly if classes **Monster** and **CookieMonster** appear in the second pane. More to our point, you should also see a class template in the **bottom** pane, awaiting your entry for a new class.

Create two new classes, **Frog** and **Grouch**.

* These should each be **subclasses** of class **Monster**.
* Each should have an **initialization** method category with an **initialize** method that explicitly calls **Monster**'s version of the **initialize** method, but then gives a different greeting.

  A **Frog** instance's greeting should be **'Hi ho'**; and **Grouch** instance's greeting should be **'Go away'**

* Test these new classes-thus-far with the following code (in a Workspace and Transcript):

```
| kermit oscar |
kermit := Frog new initialize.
oscar := Grouch new initialize.
```

```
kermit name: 'Kermit the Frog'.
oscar name: 'Oscar the Grouch'.
kermit greet.
oscar greet.
```

* Now, let's make Monsters interact. Write a method in the **Monster** class that:
    * Takes another Monster instance as an argument.
    * Reads that monster's name.
    * Greets the monster by name.

    To the previous Workspace examples, add the following:

    ```
    kermit greetByName: oscar.
    oscar greetByName: kermit.
    ```

    ...you should see (in the Transcript):

    ```
    Hi ho, Oscar the Grouch! I am Kermit the Frog!
    Go away, Kermit the Frog! I am Oscar the Grouch!
    ```

When you are happy with these new classes, indeed run specified code above within a Workspace (and have a Transcript open to display any greetings shown). (You may also add additional testing if you'd like.) Save the resulting Workspace and Transcript as **ws4.txt** and **ts4.txt**, to be submitted.

Problem 5

You now have these three subclasses of class Monster. Expand on them in some interesting way.

* It needs to be significant -- adding methods at least, perhaps also adding additional classes, if inspiration moves you. (Adding methods or capabilities to class Monster is also okay!)
* **All three Monster subclasses** need to be **modified** in some appropriate fashion, by the time you are done; at least **six additional methods** need to be **added** to the three Monster subclasses and/or class Monster. Include comments in your new methods (and classes, if any).

  Perhaps each Monster subclass can have more capabilities; these might be more interesting interactions, or something graphical, or even something sound-based. (Obviously you'd have to do some research to add graphical or sound-based capabilities.)
* Make sure that all of your new code is within the SesameStreet class category.
* Open a Workspace and Transcript, and include code that tests/demonstrates your new capabilities; save their contents as **ws5.txt** and **ts5.txt**, and submit them.

When you are done, you need to **export** the final version of all of your code within the **SesameStreet** class category (you need to perform a **fileOut**). In a System Browser, click on **SesameStreet** in the first pane, and yellow-click, selecting **fileOut**. It will note that SesameStreet.st already exists; select **choose another name**, and save your code under the new file name **SesameStreet2.st**, and **accept**.

Submit your files of **SesameStreet.st** and **SesameStreet2.st**, as well as all of the saved Workspace and Transcript files specified.