In this exercise, you will write your first complete program. The program is very simple, the goal of the exercise is to get used to the Pharo environment and the basics of the language. From the *Pharo by Example* book, you should read the chapters 5-7.

Use a fresh pharo1.0-10451-BETAweb09.09.3 image to answer all questions.

3. A Simple Counter

We want you to implement a simple counter that follows the small example given below.

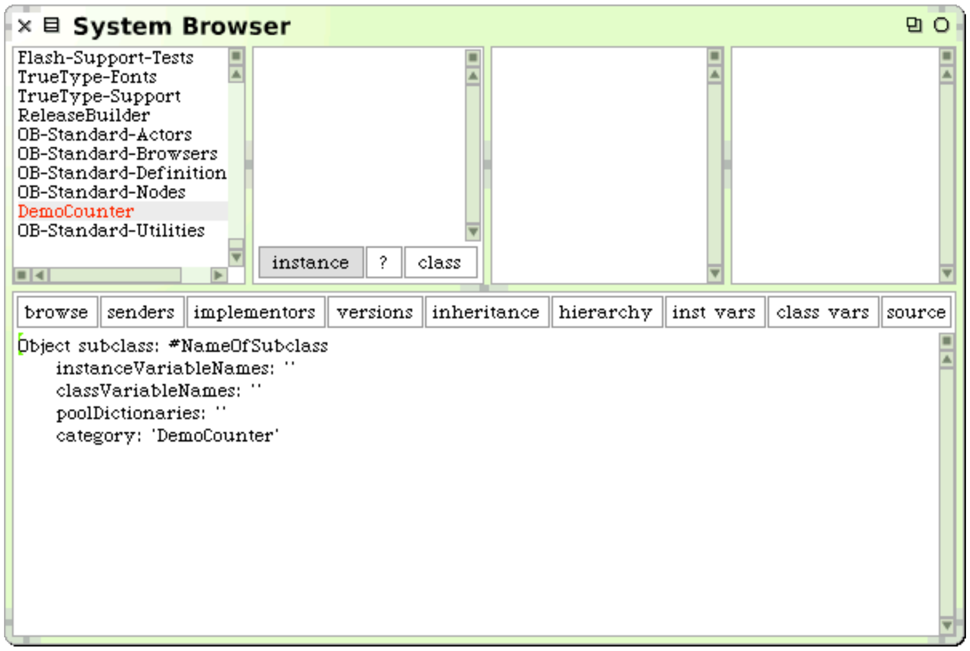
```
|counter|
counter := SimpleCounter new.
counter increment; increment.
counter decrement.
counter value = 1
```

The first part of this exercise series is a tutorial that will guide you in building the counter example.

Creating your own class

In this part you will create your first class. A class is associated with a category (a folder containing the classes of your project).

The steps we will do are the same ones every time you create a class, so memorize them well. We are going to create a class `SimpleCounter` in a category called `DemoCounter`. Figure 1 shows the result of creating such a category.



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

Figure 1: The category is created.

Creating a Class category

In the System Browser, get the context menu on the left pane (alt-click on the mac) and select *add item*. The system will ask you a name. You should write `DemoCounter`. This new category will be created and added to the list.

Creating a Class

Creating a class requires five steps. They consist basically of editing the class definition template to specify the class you want to create.

1. **Superclass Specification**. First, specify the superclass of the class you are creating. The class definition template should show `Object` by default, which is exactly what we want.

2. **Class Name**. Next, you should fill in the name of your class by replacing the word `NameOfSubClass` with the word `SimpleCounter`. Take care that the name of the class starts with a capital letter and that you do not remove the # sign in front.

3. **Instance Variable Specification**. Then, you should fill in the names of the instance variables of this class. We need one instance variable called `value`.

4. **Class Variable Specification**. As we do not need any class variable make sure that the argument for the class instance variables is an empty string (`classVariableNames: ''`).

5. **Compilation**. That's it! We now have a filled-in class definition for the class `SimpleCounter`. To define it, we still have to **compile** it. Therefore, select the **accept** option from the operate menu (right-click button of the mouse). The class `SimpleCounter` is now compiled and immediately added to the system.

As we are disciplined developers, we provide a comment to `SimpleCounter` class by clicking **?** button of the class definition. You can write the following comment:

```
SimpleCounter is a concrete class which supports incrementing
and decrementing a counter.

Instance Variables:

value        <Integer>
```

Select **accept** to store this class comment in the class.

Defining protocols and methods

In this part you will use the System Browser to learn how to add protocols and methods.

Creating and Testing methods

The class we have defined has one instance variable `value`. You should remember that in Smalltalk, everything is an object, that instance variables are private to the object and that the only way to interact with an object is by sending messages to it.

Therefore, there is no other mechanism to access the instance variables from outside an object than sending a message to the object. What you can do is to define messages that return the value of the instance variable of a class. Such methods are called **accessors**, and it is a common practice to always define and use them. We start to create an accessor method for our instance variable `value`.

Remember that every method belongs to a protocol. These protocols are just a group of methods without any language semantics, but convey important navigation information for the reader of your class. Although protocols can have any name, Smalltalk programmers follow certain conventions for naming these protocols. If you define a method and are not sure what protocol it should be in, first go through existing code and try to find a fitting name.

An important remark: *Accessors* are normally defined in protocols `accessing` or `private`. Use the `accessing` protocol when a client object (like an interface) really needs to access your data. Use `private` to clearly state that no client should use the accessor. This is purely a convention. There is no way in Smalltalk to enforce access rights like *private* in C++ or Java. To emphasize that objects are not just data structure but provide services that are more elaborated than just accessing data, put your accessors in a `private` protocol. As a good practice, if you are not sure then define your accessors in a `private` protocol and once some clients really need access, create a protocol `accessing` and move your methods there. Note that this discussion does not seem to be very important in the context of this specific simple example. However, this question is central to the notion of object and encapsulation of the data. An important side effect of this discussion is that you should always ask yourself when you, as a client of an object, are using an accessor if the object is really well defined and if it does not need extra functionality.

Exercise 3.1: Accessor

Decide in which protocol you are going to put the accessor for `value`. We now create the accessor method for the instance variable `value`. Start by selecting the class `DemoCounter` in a browser, and make sure the **Instance** button is selected. Create a new protocol by clicking the right-button of the mouse on the pane of methods categories, and choosing `New Category`, and give a name. Select the newly created protocol. Then in the bottom pane, the edit field displays a method template laying out the default structure of a method. Replace the template with the following method definition:

```
value
  "return the current value of the value instance variable"

  ^value
```

This defines a method called `value`, taking no arguments, having a method comment and returning the instance variable `value`. Then choose **accept** in the operate menu (right button of the mouse) to compile the method. You can now test your new method by typing and evaluating the next expression in a Workspace, in the Transcript, or any text editor `SimpleCounter new value`.

This expression first creates a new instance of `SimpleCounter`, and then sends the message `value` to it and retrieves the current value of `value`. This should return `nil` (the default value for noninitialised instance variables; afterwards we will create instances where `value` has a reasonable default initialisation value).

Exercise 3.2: Mutator

Another method that is normally used besides the *accessor* method is a so-called *mutator* method. Such a method is used to *change* the value of an instance variable from a client. For example, the next expression first creates a new `SimpleCounter` instance and then sets the value of `value` to 7:

```
SimpleCounter new value: 7
```

This mutator method does not currently exist, so as an exercise write the method `value:` such that, when invoked on an instance of `SimpleCouter`, the `value` instance variable is set to the argument given to the message. Test your method by typing and evaluating the expression above.

Exercise 3.3: Operations

Implement the following methods in the protocol `operations`.

```
increment
    self value: self value + 1
decrement
    self value: self value - 1
```

Now test the methods `increment` and `decrement` but pay attention that the counter value is not initialized. Try:

`SimpleCounter new value:  0; increment; value.`

Exercise 3.4: Printing

Implement the following methods in the protocol `printing`

```
printOn: aStream
    super printOn: aStream.
    aStream nextPutAll: ’ with value: ’,
    self  value printString.
    aStream cr.
```

Now test the printing method by evaluating the following code:

`SimpleCounter new value:  5.`

Note that the method `printOn:` is used when you print an object or click on `self` in an inspector.

Exercise 3.5: Adding an instance initialization method

Now we have to write an initialization method that sets a default value to the `value` instance variable. However, as we mentioned the `initialize` message is sent to the newly created instance. This means that the `initialize` method should be defined at the instance side as any method that is sent to an instance of `SimpleCounter` like `increment` and `decrement`. The `initialize` method does not have specific and predefined semantics; it is just a convention to name the method that is responsible to set up the instance variable default values.

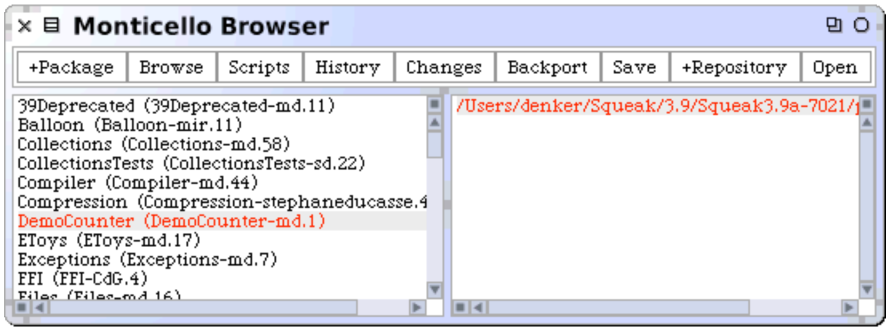
Therefore at the instance side, you should create a protocol `initialization`, and create the following method (the body of this method is left blank. Fill it in!).

```
initialize
   "set the initial value of the value to 0"
```

Now create a new instance of class `SimpleCounter`. Is it initialized by default? The following code should now work without problem: `SimpleCounter new increment`

Exercise 3.6: Monticello Package

While going through the tutorial, you have build code for a class that's inside a category. Now we want to save this code with Monticello. For that, open the Monticello Browser, add a package for DemoCounter and save it on your disk (or SqueakSource).



```
39Deprecated (39Deprecated-md.11)
Balloon (Balloon-mir.11)
Collections (Collections-md.58)
CollectionsTests (CollectionsTests-sd.22)
Compiler (Compiler-md.44)
Compression (Compression-stephaneducasse.4
DemoCounter (DemoCounter-md.1)
EToys (EToys-md.17)
Exceptions (Exceptions-md.7)
FFI (FFI-CdG.4)
Files (Files-md.16)
```

```
/Users/denker/Squeak/3.9/Squeak3.9a-7021/
```

Figure 2: Your package is created.

Exercise 3.7: SUnit

In the tutorial, we tested a lot of code by hand. Wouldn't it be nice to be able to recored those tests for easy replay? That's what Unit Tests are made for.

First add a class SimpleCounterTest:

```
TestCase subclass: #SimpleCounterTest
instanceVariableNames: ''
classVariableNames: ''
poolDictionaries: ''
category: 'DemoCounter'
```

Then, to make sure everything works, a trivial test method:

```
testNew
self assert: SimpleCounter new class = SimpleCounter
```

Open the testrunner and run the test to see if everything is Ok!

At the beginning of the tutorial, we had a snipped of code:

```
|counter|
counter := SimpleCounter new.
counter increment; increment.
counter decrement.
counter value = 1
```

Take this code and make a test out of it.

Write one test for everything tested by hand before (that is, one test for exercises 3.1 to 3.5 each).

Exercise 3.8: Another instance creation method

If you want to be sure that you have really understood the distinction between instance and class methods, you should now define a different instance creation method named `withValue:`. This method receives an integer as argument and returns an instance of `SimpleCounter` with the specified value. The following expression should return 20.

```
(SimpleCounter withValue: 19) increment ; value
```

- Before writing any code, write a test!
- Now implement the method `withValue:` to get the test green.

Exercise 3.9: Collections

```
| anArray sum |
sum := 0.
anArray := #(21 23 53 66 87).
anArray do: [:item | sum := sum + item].
sum
```

What is the final result of sum ? How could this piece of code be rewritten to use explicit array indexing (with the method `at:`) to access the array elements[1]? Test your version. Rewrite this code using inject:into:

Please save the Monticello package `DemoCounter` and send it by mail to st-staff@iam.unibe.ch. Attach your written solutions that are not part of the source-code to the mail or hand them in as hardcopy at the beginning of the next exercise session. Your mail and solutions should be clearly marked with names and matrikel numbers of all solution authors.

[1]Note this is how you would implement it with Java or C++