

Developing a simple counter

To get started in Pharo, we invite you to implement a simple counter by following the steps given below. In this exercise you will learn how to create packages classes, method, instances. You will learn how to define tests and more. This simple tutorial covers most of the important actions that we do when developing in Pharo.

Note that the development flow promoted by this little tutorial is *traditional* in the sense that you will define a package, a class, *then* define its instance variable *then* define its methods *and* finally execute it. The companion video follows also such programming development flow. Now in Pharo, developers often follows a *totally* different style (that we call live coding) where they execute an expression that raises errors and they code in the debugger and let the system define some instance variables and methods on the fly for them. Once you will have finish this tutorial, you will feel more confident with Pharo and we strongly suggest you to try the other style by following the second video showing such different development practices.

1.1 Our use case

Here is our use case: We want to be able to create a counter, increment it twice, decrement it and check that its value is correct. It looks like this little use case will fit perfectly a unit test - you will define one later.

```
| counter |  
counter := Counter new.  
counter increment; increment.  
counter decrement.  
counter count = 1
```

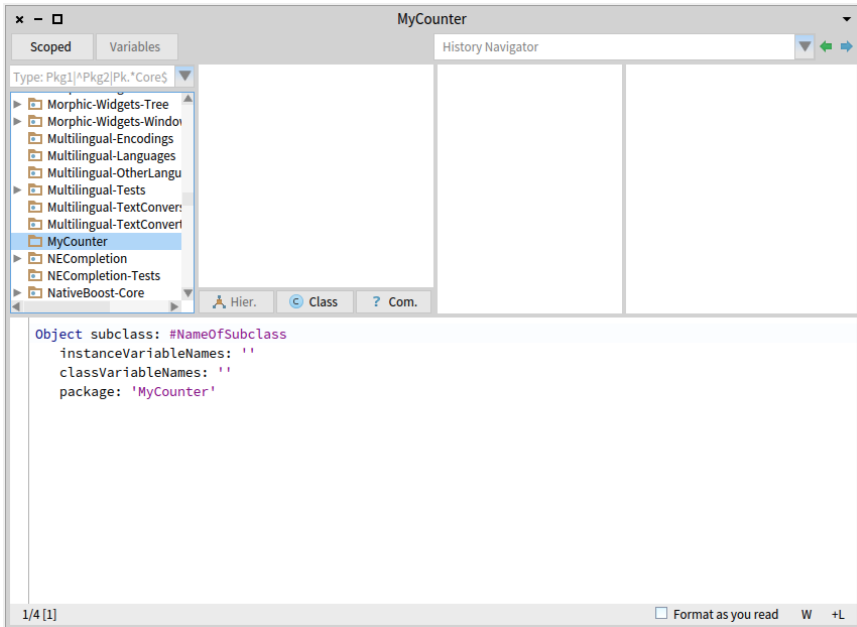


Figure 1.1 Package created and class creation template.

Now we will develop all the mandatory class and methods to support this scenario.

1.2 Create your own class

In this part, you will create your first class. In Pharo, a class is defined in a package. You will create a package then a class. The steps we will do are the same ones every time you create a class, so memorize them well.

Create a package

Using the Browser create a package. The system will ask you a name, write Counter. This new package is then created and added to the list. Figure 1.1 shows the result of creating such a package.

Create a class

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

- Superclass Specification. First, you should replace the word `NameOfSubclass` with the word `Object`. Thus, you specify the superclass of

the class you are creating. Note that this is not always the case that `Object` is the superclass, since you may inherit behavior from a class specializing already `Object`.

- **Class Name.** Next, you should fill in the name of your class by replacing the word `NameOfClass` with the word `Counter`. Take care that the name of the class starts with a capital letter and that you do not remove the `#` sign in front of `NameOfClass`. This is because the class we want to create does not exist yet, so we have to give its name, and we use a `Symbol` (a unique string in `Pharo`) to do so.
- **Instance Variable Specification.** Then, you should fill in the names of the instance variables of this class. We need one instance variable called `count`. Take care that you leave the string quotes!
- **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 `classInstanceVariableNames: ''`.

You should get the following class definition.

```
Object subclass: #Counter
  instanceVariableNames: 'count'
  classVariableNames: ''
  package: 'MyCounter'
```

Now we should compile it. We now have a filled-in class definition for the class `Counter`. To define it, we still have to *compile* it. Therefore, select the accept menu item. The class `Counter` is now compiled and immediately added to the system.

Figure 1.2 illustrates the resulting situation that the browser should show.

The tool runs automatically some code critic and some of them are just inaccurate, so do not care for now.

As we are disciplined developers, we add a comment to `Counter` class by clicking `Comment` button. You can write the following comment:

```
Counter is a simple concrete class which supports incrementing and
  decrementing a counter.
Its API is
- decrement, increment
- count
Its creation API is message withValue:
```

Select menu item 'accept' to store this class comment in the class.

1.3 Define protocols and methods

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

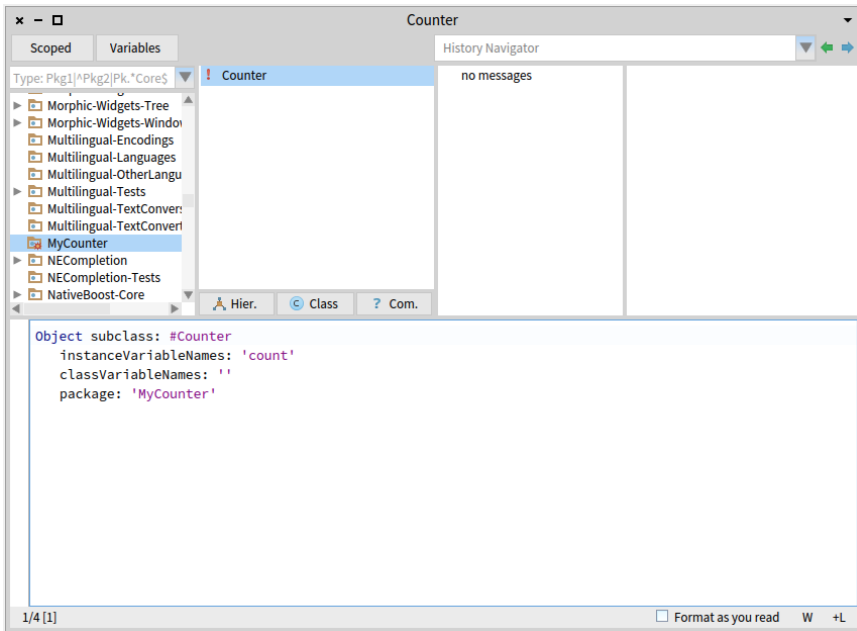


Figure 1.2 Class created.

The class we have defined has one instance variable named `count`. You should remember that in Pharo, (1) everything is an object, (2) that instance variables are private to the object, and (3) 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 variable values 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. 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 `count`.

A method is usually sorted into 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, Pharo 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.

Create a method

Now let us create the accessor methods for the instance variable `count`. Start by selecting the class `Counter` in a browser, and make sure the you are edit-

ing the instance side of the class (i.e., we define methods that will be sent to instances) by deselecting the Class side radio button.

Create a new protocol by bringing the menu of methods protocol list. 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. As a general hint, double click at the end of or beginning of the text and start typing your method. Replace the template with the following method definition:

```
[ count
  "return the current value of the value instance variable"
  ^ count
```

This defines a method called `count`, taking no arguments, having a method comment and returning the instance variable `count`. Then choose *accept* in the menu to compile the method. You can now test your new method by typing and evaluating the next expression in a Playground, or any text editor.

```
[ Counter new count
  > nil
```

This expression first creates a new instance of `Counter`, and then sends the message `count` to it. It retrieves the current value of the counter. This should return `nil` (the default value for non-initialised instance variables). Afterwards we will create instances with a reasonable default initialisation value.

Adding a setter method

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

The snippets shows that the counter effectively contains its value.

```
[ | c |
  c := Counter new count: 7.
  c count
  > 7
```

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

1.4 Define a Test Class

Writing tests is an important activity that will support the evolution of your application. Remember that a test is written *once and executed million* times.

For example if we have turned the expression above into a test we could have checked automatically that our new method is correctly working.

To define a test case we will define a class that inherits from `TestCase`. Therefore define a class named `CounterTest` as follows:

```
TestCase subclass: #CounterTest
  instanceVariableNames: ''
  classVariableNames: ''
  package: 'Counter'
```

Now we can write a first test by defining one method. Test methods should start with *text* to be automatically executed by the `TestRunner` or when you press on the icon of the method. Now to make sure that you understand in which class we define the method we prefix the method body with the class name and `>>`. `CounterTest>>` means that the method is defined in the class `CounterTest`.

Define the following method. It first creates an instance, sets its value and verifies that the value is correct. The message `assert:` is a special message verifying if the test passed or not.

```
CounterTest >> testCountIsSetAndRead
| c |
c := Counter new.
c count: 7.
self assert: c count = 7
```

Verify that the test passes by executing either pressing the icon in front of the method or using the `TestRunner` available in the `Tools` menus (selecting your package). Since you have a first green test. This is a good moment to save your work.

1.5 Saving your work

Several ways to save your work exist.

- *Using plain files.* You can save the class or a method by clicking on it and selecting the `fileout` menu item. You will get a file containing the source code on your hard-disc - This is not the favorite way to save your code.
- *Using a version control system.* It is better to use a version control system. In Pharo you can use `Monticello` and `Git` (even if it is more for advanced users).

In this tutorial we explain the simplest way to get you done. Note that the complete set of Pharo packages is managed via `Monticello` (which is a distributed versioning control system - there is are chapters in **Pharo by Example** and **Deep into Pharo** books <http://books.pharo.org>).

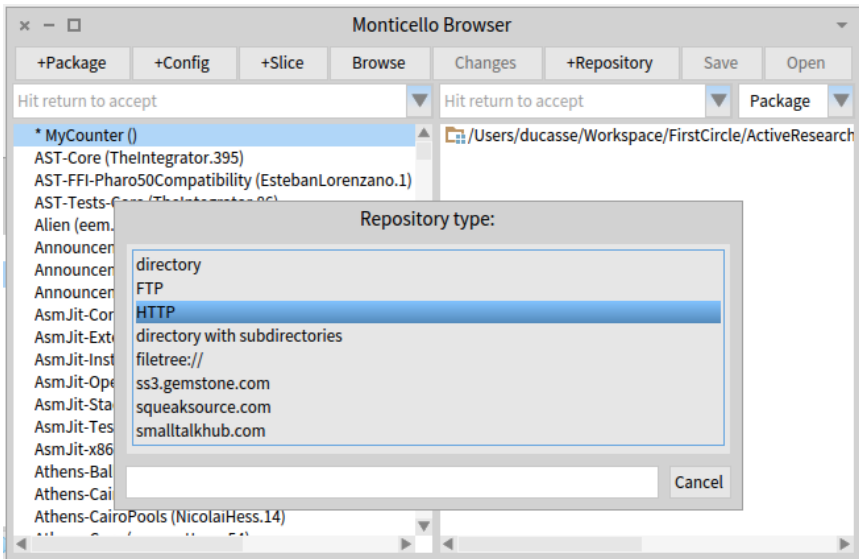


Figure 1.3 Selecting a new kind of repository to the list of possible places to commit the package.

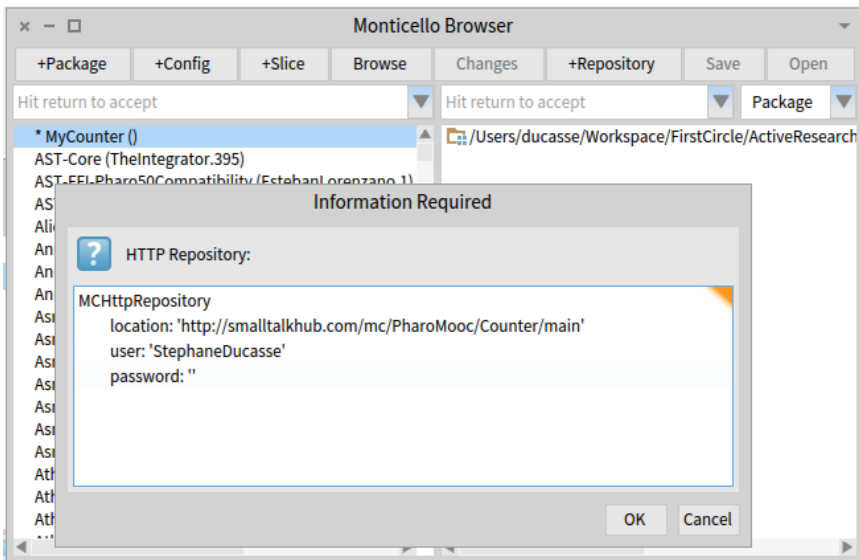


Figure 1.4 Editing the repository information.

Use the **Monticello Browser** (available in Tools) to save your work. You can save a package locally on your harddisc or on a remote server on the web such <http://www.smalltalkhub.com>

Saving using Monticello

Using Monticello you can save your work:

- *Locally*. You can store your packages in a folder on your disc (use directory as a kind of repository).
- *Remotely*. Using an account on a free server such <http://www.smalltalkhub.com/>. You can save your work and share it with others.

Note each time you load or save a package, this package is also be stored on the folder named 'package-cache' on your hard-disc.

Add a repository.

Go to <http://www.smalltalkhub.com/> and create a member account then register a new project. You get an HTTP entry that refers to your project. Define a new HTTP repository using the Monticello Browser as shown by Figures 1.3 and 1.4.

Figure 1.3 shows that you package is dirty: this is indicated with the little '*' in front of the packages.

Example. We are saving our examples into a special team named PharoMooC in the the Counter project so the template is the following one

```
MCHttpRepository
  location: 'http://smalltalkhub.com/mc/PharoMooC/Counter/main'
  user: ''
  password: ''
```

So the template is the following:

```
MCHttpRepository
  location:
    'http://smalltalkhub.com/mc/YourAccountOrTeamAccount/YourProject/main'
  user: 'YourAccountID'
  password: 'YourAccountPassword'
```

Saving your package.

To save your work, simply select your package and the repository you want to save it to and save it using the Save button. This will open a dialog where you can give a comment, version numbers and blessing. From then on, other people can load it from there, in the same way that you would use cvs or other multi-user versioning systems. Saving the image is also a way to save

your working environment, but not a way to version and publish it in a way that can be easily shared.

You can of course both publish your package (so that other people can load it, and that you can compare it with other versions, etc.) and save your image (so that next time that you start your image you are in the same working environment).

1.6 Adding more messages

Before implementing the following messages we define first a test. We define one test for the method `increment` as follows:

```
CounterTest >> testIncrement
| c |
c := Counter new.
c count: 0 ; increment; increment.
self assert: c count = 2
```

- Propose a definition for the method `increment`.
- Define a test and method for the method `decrement`.
- Implement the following methods `increment` and `decrement` in the protocol 'operation'.
- Implement also a new test method for the method `decrement`

```
Counter >> increment
self count: self count + 1
```

```
Counter >> decrement
self count: self count - 1
```

Run your tests they should pass (as shown in Figure 1.5). Again this is a good moment to save your work. Saving at point where tests are green is always a good process.

1.7 Better object description

When you open an `inspect` (putting a `self halt` inside a method definition) you obtain an inspector or when you select the expression `Counter new` and print its result (using the `Print it` menu of the editor) you obtain a simple string 'a Counter'.

```
Counter new
> a Counter
```

We would like to get a much richer information for example knowing the counter value. Implement the following methods in the protocol `printing`

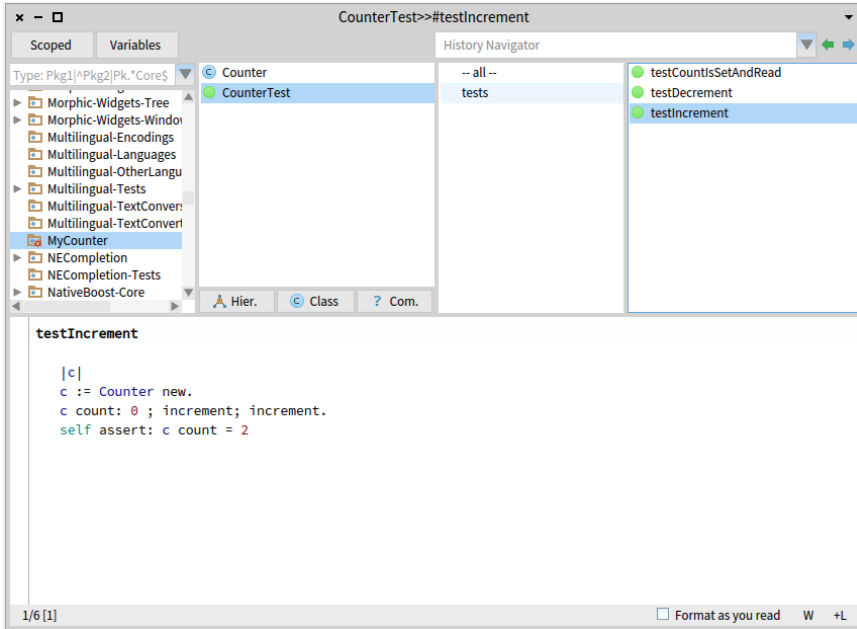


Figure 1.5 Class with green tests.

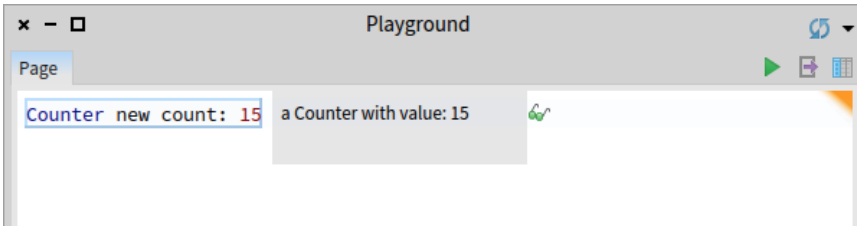


Figure 1.6 Better description.

```
Counter >> printOn: aStream
  super printOn: aStream.
  aStream nextPutAll: ' with value: ', self count printString.
```

Note that the method `printOn:` is used when you print an object using `print` it (See Figure 1.6) or click on `self` in an inspector.

1.8 Instance initialization method

Right now the initial value of our counter is not set as the following expression shows it.

1.9 Define a new instance creation method

```
[ Counter new count  
  > nil
```

Let us write a test checking that a newly created instance has 0 as a default value.

```
[ CounterTest >> testValueAtCreationTimeIsZero  
  self assert: Counter new count = 0
```

If you run it, it will turn yellow indicating a failure (a situation that you anticipated but that is not correct) - by opposition to an error which is an anticipated situation leading to failed assertion.

Define an initialize method

Now we have to write an initialization method that sets a default value of the count 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 `Counter` (like `increment`) and `decrement`. The `initialize` method 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!).

```
[ Counter >> initialize  
  "set the initial value of the value to 0"
```

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

```
[ Counter new increment
```

and the following one should return 2

```
[ Counter new increment; increment; count  
  > 2
```

Again save your work before starting the next step.

1.9 Define a new instance creation method

We would like to show you the difference between an instance method (i.e. sent to instances) and a class method (i.e. to a class). In fact the only difference is the place to define them. An instance method is defined in the instance side of Code Browser while class methods are defined on the class side (Pressing the button `Class`).

Define a different instance creation method named `withValue:`. This method receives an integer as argument and returns an instance of `Counter` with the specified value.

Let us define a test:

```
CounterTest >> testAlternateCreationMethod
  self assert: ((Counter withValue: 19) increment ; count) = 20
```

Here the message `withValue:` is sent to the class `Counter` itself.

Your implementation should look like

```
Counter class >> withValue: anInteger
  | c |
  c := self new count: anInteger.
  ^ c
```

Note that `self` in such method refers to the class `Counter` itself.

1.10 Conclusion

In this tutorial you learned how to define packages, classes, methods, and define tests. The flow of programming that we chose for this first tutorial is similar to most of programming languages. In Pharo you can use a different flow that is based on defining a test first, executing it and when the execution raises error to define the corresponding classes, methods, and instance variable often from inside the debugger. We suggest you now to redo the exercise following the second companion video.