

SCHOOL OF COMPUTER SCIENCE



PRIFYSGOL
BANGOR
UNIVERSITY

Object Oriented Programming in Java

Laboratory 2 Introducing Classes

Dave Perkins

Introduction

In this laboratory session you will learn how to:

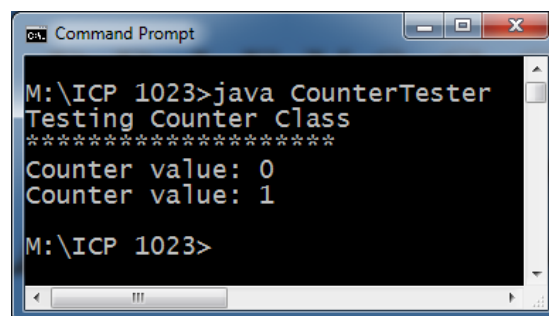
- Design and implement a range of simple classes;
- Create instances of the class you have defined;
- Use *mutator* and *access* methods;
- Use *static* and *non-static* methods.

Before working through these notes have a quick look at the lecture slides (Lectures 1 and 2) which introduce the topic of classes and objects. Use the code in the lecture slides to help you develop a solution to the assessed laboratory work in these notes. Also, keep a copy of the course text besides you for reference purposes.

These notes contain assessed laboratory work.

Exercise 1: Using the Counter Class

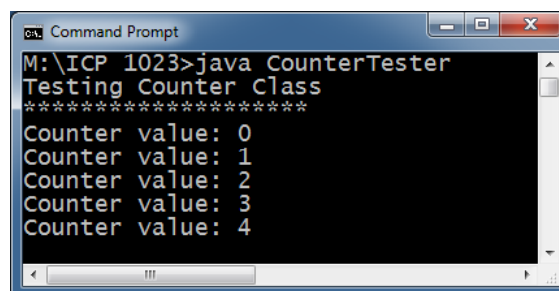
Make a copy of Horstmann's **Counter** class. Compile the Java code and remove any syntax errors. Create a second file called **CounterTester** which generates the following output:



```
Command Prompt
M:\ICP 1023>java CounterTester
Testing Counter Class
*****
Counter value: 0
Counter value: 1
M:\ICP 1023>
```

Figure 1 Testing the Counter Class

The code for **Counter** and **CounterTester** can be found in the slides for Lecture 1. Modify **CounterTester** so that the following output is generated.



```
Command Prompt
M:\ICP 1023>java CounterTester
Testing Counter Class
*****
Counter value: 0
Counter value: 1
Counter value: 2
Counter value: 3
Counter value: 4
```

Figure 2 Modified Version of CounterTester

Exercise 2: Adding Functionality to the Counter Class

Add two extra methods to the **Counter** class. The first method **reset()** sets the value of the counter to zero; the second which is called **decrement()** subtracts one from the value of the counter *unless* the counter is already set to zero in which case there is no modification of the state.

Having provided this extra functionality you will need to recompile **Counter.java** to obtain an updated class file. In order to test that these two new methods work it will be necessary to amend **CounterTester.java** so that both methods are invoked at least once.

Make sure that your test program generates appropriate test narrative.

Exercise 3: Using The Bank Account Class

By tradition the **BankAccount** class is often the first class that students encounter on a Java course. In today's laboratory session a version of this class will be implemented and tested. Although most of the code will be provided for you, do not simply copy and paste but rather spend some time studying the code and understanding how it works.

To get started on this exercise use **Appendix 1** to create a file called **BankAccount.java**. Note that the code is incomplete and that you will need to replace the question marks by appropriate Java expressions. The code supplied also contains deliberate syntax errors. Compile the source file and remove these errors.

Using the code template for **BankAccountTester** create a test driver that will

- Create two instances of **BankAccount** – the first instance should be called **jacksAccount** and the second instance will be called **jillsAccount** (Jack and Jill have accounts at the same bank)
- Open Jack's account with a balance of zero pounds whilst Jill's account is opened with a deposit of £500
- Make a deposit of £200 into Jack's account
- Make a withdrawal of £200 from Jill's account
- Make a withdrawal of £500 from Jack's account
- Use an appropriate method to display the final balance of each account

Check that the value of both balances is correct. Make sure that you perform a calculation with respect to each balance before you actually run the program. These two expected values constitute test data.

Exercise 4: The Credit Crunch Comes

As you can see there is a problem with our bank account system because it allows reckless individuals like Jack to make withdrawals even when they have no money in their account.

Solve this problem by altering the implementation of the `withdraw()` method. The new implementation allows account holders to withdraw as much as they want but *only* whilst their balance remains greater than or equal to zero. Any attempt withdraw more than the existing balance results in the balance being set to zero. Do not insert any error messages or warnings into the `withdraw()` method.

Use your test driver to check that the amended version of `withdraw()` works as expected.

Finally, write a `toString()` method for the class (see Horstmann's *Big Java:Late Objects*, p.446).

This method will return a string representation of the object's state. So for a `BankAccount` object the string returned will look like this:

```
BankAccount[balance = 500] // See note below
```

The header for the method is listed below:

```
public String toString()
```

Because the class has been modified it should now be tested again. Revise `BankAccountTester` so that it includes test code for the two new methods which have just been supplied.

Finally, read Horstmann on the topic of *unit testing*.

Addendum

If `BankAccount` contained another instance variable e.g. `accountNumber` then the `toString` method should create the following string:

```
BankAccount[balance = 500, accountNumber = 734089427]
```

Assessed Laboratory Work

Developing the Person Class

For this piece of laboratory work, you are required to implement a **Person** class that could be used to hold data representing individual people. The class should store the following information with respect to each person:

- forename
- surname
- age
- height (in metres)
- gender

Each piece of information must be stored using **private** variables of appropriate types. As the access type of these instance variables is **private**, appropriate **get** and **set** methods for each variable should be provided.

Write a **toString()** method to display the contents of each instance variable. A sample return string for this method looks like the following:

Person[foreName=joe, surName=smith, age=25, height=1.57, gender=male]

Also write a **format()** method which constructs a string containing formatting information such that when the string is printed the data values are neatly aligned in columns and thus suitable for screen display similar to the following

smith	joe	25	1.57	male
davis	sian	18	1.73	female
...

To see how to write a **format** method look at Horstmann's invoicing case study in Chapter 11, *Big Java*. Note that to write this method you are required to use the **format()** method in the class **String**.

Make sure you understand the difference between a **toString()** and a **format()** method; if you are not sure ask the lecturer or one of the demonstrators.

Save your source code in a file called **Person.java**. For a description of what a **toString** method should do, see Horstmann's *Big Java: Late Objects* (p.446). Note also that both methods return an object of type **String**.

Make sure that all methods are provided with Javadoc style comments.

Testing the Person Class

Write a test driver called **PersonTester.java** which tests all of the methods in your class. This class must include a **main** method to allow execution of the program. Do *not* supply test values at run-time but follow the approach presented in the lecture slides with respect to test drivers. Also provide test narrative in your output.

Adding a Static Variable and Method

The next task is to add a **static** variable and method to the class **Person**. The **private static** variable is called **personCount** and should store the number of persons that have been created by the test driver. As **static** variables and methods are associated with the *class* and not the instances of the class, there will only ever be one copy of the **static** variable **personCount**. This variable should be incremented by an appropriate instruction in the class constructor so that whenever a new instance of the class is created the **personCount** variable is incremented.

For the final part of this laboratory, you are required to examine the behaviour of the additional variable that has been added to the class under development. To view the value of **personCount** add a static method to **Person** class called **getPersonCount()**. This method simply returns the value of **personCount** to the test driver.

Although static variables have not been covered in the lecture course this topic is easily mastered. See the Oracle tutorial site for more information:

<http://docs.oracle.com/javase/tutorial/java/javaOO/classvars.html>

By contrast, the use of static methods has been discussed and method invocation should follow the pattern below:

ClassName.methodName; e.g. **Math.pow(2,3)** ;

Your main method in **PersonTester** should create five instances (i.e. objects) of your **Person** class and then display their details together with the final value of **personCount** which should hold the value five.

The details of the five persons **MUST NOT** be read from the keyboard at run-time but instead supplied through the test driver as parameters to an appropriate constructor.

Note that **static** variables are sometimes called *class* variables because they are shared by all instances of the class in question.

Submission Notes and Marking Scheme

Use **Blackboard** to submit your source code files. The deadline for submission will be published on Blackboard. Late submissions will be penalised in line with School policy.

Marks for this laboratory exercise are awarded as follows:

- Correct implementation of the class **Person**
 - Instance variables 10%
 - Getters and setters 10%
 - toString 15%
 - format 15%
- Effective testing strategy 25%
- Correct implementation of static features
 - Static variable 5%
 - Static method 10%
- Comments, layout and structure 10%

Students who submit work but have a poor understanding of what has been submitted may be *heavily penalised*. When making a submission it is your responsibility to ensure that all code submitted via Blackboard is:

- Consistent with stated requirements
- Entirely your own work
- Submitted through Blackboard on time

Please note that there are **severe penalties** for submitting work which is not your own. *If you have used code which you have found on the Internet or from any other source* then you **must** signal that fact with appropriate program comments.

Note also that to obtain a mark you *must attend a laboratory session* and be prepared to demonstrate your program and answer questions about the coding. Non-attendance at labs will result in your work **not** being marked.

Steve Marriott

Appendix 1: BankAccount

```
/**
 * A bank account has a balance that can be changed by
 * deposits and withdrawals.
 * WARNING: CODE MAY DELIBERATELY CONTAIN ERRORS
 * WARNING: CODE IS INCOMPLETE
 */

public class BankAccount
{
    // Constructs a bank account with a zero balance.
    public void BankAccount()
    {
        balance = 0;
    }

    // Constructs a bank account with a given balance.
    public BankAccount(double initial Balance)
    {
        balance = ???;
    }

    // Deposits money into the bank account.
    public void deposit(double amount)
    {
        double newBalance = balance + amount;
        balance = newBalance;
    }

    // Withdraws money from the bank account.
    public void withdraw(double amount)
    {
        double newBalance= ???;
        balance = ???;
    }

    // Gets the current balance of the bank account.
    public double getBalance()
    {
        return ???;
    }

    private double balance;
}
```



```
/**
    A class to test the BankAccount class.
 */

public class BankAccountTester
{
    // Tests the methods of the BankAccount class.

    public static void main(String[] args)
    {
        BankAccountjacksAccount = new BankAccount();

        // to be completed
    }
}
```