SCHOOL OF COMPUTER SCIENCE

PRIFYSGOL
BANGOR
UNIVERSITY

Object Oriented Programming in Java

Laboratory 3
Building and Testing Classes

Dave Perkins

Introduction

This laboratory session provides more practice in developing simple classes from informal natural language (i.e. English) descriptions . We also continue to emphasise the necessity *of thoroughly testing* all of the methods in any class that we implement.

Note that some *standard forms* to assist test documentation are provided in the notes and you should make use of these forms in documenting your own programs.

Exercise 1: The Product Class

Implement a class **`Product`** using appropriate data types for the instance variables. For the purposes of this exercise assume that a product has both a *description* and a *price*.Thus a toaster costing £29.95 would be constructed as follows:

```
Product aProduct=  new Product("Toaster", 29.95);
```

You are required to supply the following methods:

- **`getDescription()`** – returns a description of the product
- **`getPrice()`** – returns the price of the product
- **`reducePrice()`** – reduces the price by a specified percentage
- **`increasePrice()`** – increases the price by a specified percentage
- **`toString()`** – returns product details in the form of a single string
- **`format()`** – returns product details as a *formatted* string

When you have compiled this class and it is free of syntax errors write a test driver to test *all* of the methods listed above. Make sure that the driver generates test narrative.

The test driver should be stored in a file called **`ProductTester.java`**.

See Horstmann's *Big Java:Late Objects*, (pp.570-573) for examples of format methods.

Exercise 2: The Student Class

Implement a class **`Student`**. Every student has

- A forename;
- A surname;
- An alphanumeric identifier of length eight (100256UG);
- A degree course (e.g. B.Sc. Computer Science, B.A. History);

Choose appropriate instance variables then provide one or more constructors for this class together with all appropriate get and set methods. Also provide a **`toString()`** method which returns a string representation of a student and a **`format()`** method. Design and implement a test driver for the class **`Student`**. The driver should be stored in a file called **`StudentTester.java`**

Exercise 3: The Circle Class

Write a **`Circle`** class which has a single attribute

- radius declared as a double

The class should have the following constructors and methods:

- Constructor which has no arguments and sets the radius to 0.0.
- Constructor which accepts the radius of the circle as an argument.
- **`getRadius()`** - an accessor method for the radius field.
- **`calcArea()`** - calaculates the area of the circle.
- **`calcDiameter()`** – calculates the diameter of the circle.
- **`calcCiurcumference()`** - calculates the circumference of the circle.
- **`toString()`** – return string representation in canonical form.

In performing these calculations make use of the constant **`PI`** provided in the Java class **`Math`** which is located in the **`java.lang`** package.

```
public static final double PI = 3.141592653589793
```

Also look up the **`import static`** feature provided in Java 7. This is a nice bit of syntactic sugar when using static methods and variables, (See Horstmann's, *Big Java*, p.325 Specxial Topic 7.3).

Finally, write a test driver for this class and prepare a test plan in the format shown on the next page. Remember that the *test plan* identifies the purpose of each test.

Sample Test Documentation

Sample test documentation is shown below. The test documents consist of

- Test Plan – numbered list of test cases
- Test Log – numbered list of test runs

Each test run can be cross referenced with a test case in the test plan. Tests should not be run unless their purpose has been clearly stated.

The form below is a partial *test plan* for the **`Circle`** class

Test No	Purpose of Test
1	Test **`Circle()`** i.e. the zero parameter constructor to ensure that the radius is set to 0.0
2	Test **`Circle(double radius)`** to ensure that **`this.radius`** equals **`radius`**
3	**`...`**

The *test log* is shown below. The *test data* column specifies the list of actual parameters supplied to the constructors or methods.

Test No	Test Data	Expected Output	Actual Output
1	Construct circle with zero parameter constructor	**`Circle[radius = 0.0]`** Using **`toString()`**	As expected
2	Construct circle with parameter set to 3.0 **`Circle c = new Circle(3.0);`**	**`Circle[radius = 3.0]`** Using **`toString()`**	As expected
3	. . .		

Assessed Lab Work

Developing a Playing Card Class

Write a program to implement a class called **`PlayingCard`** which can be used to create representations of standard playing cards.

The program should be based upon the following basic assumptions: every card has a *value* and a *suit*; values are either in the range 2..10 or one of the face cards – Ace, King, Queen and Jack; and finally the standard suits are Clubs, Diamonds, Hearts and Spades.

The **`PlayingCard`** class must have the following *public* methods

- **`public int getRank()`**
- **`public int getSuit()`**
- **`public String toString()`**
- **`public String format()`**

All methods must be commented in Javadoc style.

One way of dealing with the problem of representing non-numeric face values is to assign a numerical rank to the picture cards. For example:

- Jack has the rank eleven
- Queen has the rank twelve
- King has the rank thirteen
- Ace has the rank fourteen

This means that the rank of a card can be represented by an integer and that an Ace is higher than a King which in turn is higher than a Queen which in turn is higher than a Jack. Similarly, card suits can be represented as constant integer values:

```
public final int CLUBS = 0;
```

An alternative approach is to use Enumeration Types. These are covered in Horstmann's *Big Java* (4thed), p.172.

The **`toString`** method should produce the following kind of string:

```
PlayingCard[rank=King, suit=Hearts]
```

The **`format`** method should produce strings like the following:

```
2 of Clubs, King of Diamonds, 7 of Hearts, Ace of Spades
```

Alternatively, and perhaps slightly more ambitiously, you may wish to generate strings which contain the standard suit symbols:

2♣, K♦, 7♥, A♠

The class must also contain a method called **`equals()`** as partially specified below. For more information on how to code an equals method see Horstmann's discussion of this topic in *Big Java: Late Objects* (Special Topic 9.7).

```
/**
      Tests whether this card is equal to some other card.

      @param card the card to be tested.
      @return returns true if the suit and rank of test card is
      equal to the suit and rank of this card otherwise false is
      returned.
*/
public boolean equals(Object otherObject)

{ ... }
```

Finally the class should have the following constructor:

```
/**
* Constructs a card with specified initial rank and suit
*/
public PlayingCard(int rank, int suit)

{ ... }
```

Create a test driver class called **`PlayingCardTester`** and use it to test the methods you have just defined.

Using the class **PlayingCard** develop another class called **Pack**. An outline for this class is provided below:

```
public class Pack
{
   PlayingCard[] cards = new PlayingCard[];

   /**
   * Constructs a pack of 52 cards.
   * Sorted by suit Clubs, Diamonds, Hearts, Spades.
   * Sorted ascending.
   */
   public Pack()
   {
   }

   /**
   * Shuffles cards in pack.
   */
   public void shuffle()
   {
   }

   /**
   * @return string representation of 52 card pack.
   */
   public String toString()
   {

   }
}
```

Finally develop a client class **PackTester** to test all of the methods in this new class.

Submission

Use **Blackboard** to submit your source code files together with a screen shot or shots showing Javadoc output. The screen shots can be submitted as docx files.

Contain a program header

- Contain an appropriate level of comments in Javadoc style
- Follow a consistent style of indentation
- Follow the usual Java conventions for class and variable names

The deadline for submitting your work will be specified on Blackboard. Late submissions will be penalised in line with School policy.

Marks for this laboratory exercise are awarded for:

- Correct implementation of **`PlayingCard`** and **`Pack`** classes
- Systematic and documented testing (supply a *test plan* and *log*)
- Adherence to naming conventions, comments, layout and program structure
- Conceptual understanding

When submitting work it is your responsibility to ensure that all work submitted is

- Consistent with stated requirements
- Entirely your own work
- 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 to answer questions about the coding. Non-attendance at labs will result in your work not being marked.