SCHOOL OF COMPUTER SCIENCE

PRIFYSGOL BANGOR UNIVERSITY

Object Oriented Programming in Java

Laboratory 5 Simple Dice Games

Dave Perkins

Introduction

In this laboratory session we create objects derived from classes that we have either defined ourselves or have picked up from the Java library. We shall start by defining a **`Die`** class so that we can simulate the behaviour of a standard six-faced die.

Figure 1 Typical Six Sided Dice

Having defined the **`Die`** class we shall write a test class to examine the behaviour of each of the methods in the class. Once we are confident that everything is as it should be the class will be used to construct a variety of dice games each with its own simple gaming logic.

There is assessed work associated with this laboratory.

Exercise 1: Implementing the Die Class

Create a **`Die`** class with the following instance variable, constructor and methods:

```
/*
      Simulates behaviour of a die
*/
public class Die
{
      private int faceValue;      // Current face value of die

      private int sides;          // Number of sides

      private Random generator;


      public Die(int s){}

     public void throwDie(){}

      public int getFaceValue(){}

      public String toString() {}

}
```

The code for this class should be stored in a file called **Die.java**.

The constructor in this class must perform the following tasks:

- Creates an instance of the class **`Random`**;
- Sets the instance variable **`sides`** to an appropriate value, normally six but note that the user of this class may wish to create a non-standard die (i.e. one with more than six faces);
- Initialises the instance variable f**`aceValue`** to a random integer value (see paragraph below) betweeen 1 and **`sides`** inclusive.

The method **`throwDie()`** simulates the action of throwing the die onto a flat surface, an action which results in one particular face lying uppermost. This method is a *mutator* method because it is capable of altering the underlying state of a **`Die`** object. To simulate the behaviour of a conventional die you will need to generate random values in the closed interval [1,6] using the **`Random`** class in **`java.util`**.

The **`getFaceValue()`** method is an *access* method and simply returns the face value of the die. Once all the methods have been implemented, compile **`Die.java`** and remove any syntax errors. Note that at this stage you do not have a complete Java application because the **`Die`** class does not contain a **`main`** method.

Exercise 2: Writing a Test Class

Now create a tester class called **`DieTester`** which tests the constructor and *all* of the **`Die`** methods. The test driver should simulate the behaviour of a die which is thrown *thirty six* times so you should use an appropriate iterative structure (i.e. a loop) in the test driver. When the test driver is run the output should look something like this:

```
C:\MyJava:> java DieTester
DieTestrunning ...

Test 1: Test Constructor using toString
Die[faceValue=3, sides=6]

Test 2: Test getFaceValue

Inspecting die face ...
Face value = 2

Test 3: Test throwDie
Die is being thrown ...
Face values generated 5, 6, 6, 3, 2, 1, 2, 4, ........


Test finished
```

As you will notice the output of the test program traces the behaviour of the die object and thereby increases our confidence that the class has been correctly implemented.

Exercise 3: Using the Die Class to Develop a Die Game

So far we have done very little with our **`Die`** class. The test program simply creates a single **`Die`** object and then performs one or two simple operations with it. However, the **`Die`** class can be used to produce some quite interesting games involving a single die. In the second part of these lab notes you will be shown how to develop a game involving two or more **`Die`** objects and will be challenged to produce some more elaborate games.

However, we begin by doing something relatively simple by developing the game described below

> *This is a single player game called **Six or Lose**. The die is placed on the table with some particular value showing on the die face. The player starts with a score of six. The player then throws the die. If the face is a six then the game is over and the player still has a score of six. On the other hand if the die face is not six then the player loses one point and throws again. This process continues until either the player has thrown a six or the player's score has become equal to zero. The aim of the game is to throw a six before the player's score reaches zero.*

Our task is to write a program that will simulate this game. To give you an idea of what is required some screen output for this program is presented below.

```
Starting Six or Lose ...

Die          Score
*****        *****
1            5
3            4
5            3
4            2
4            1
6            1

You've thrown a six!!! You win with a score of 1.
```

Notice that in this example of the game the player *starts* by throwing a one and so loses a point from their score. On the next turn after throwing the die and getting three the player loses one point from their score which now becomes four.

Also notice that the output of the program consists of four tasks:

```
print start message
print table header
print rows of the table
print end message
```

This list of tasks should help you to grasp the structure of a program to implement this game.

Here is the output for another game; this time note that the player loses.

```
Starting Six or Lose ...

Die face        Score
*********       *****
3               5
2               4
3               3
3               2
5               1
4               0

You have not thrown a six. You lose.
```

How is this program to be written? Well the first thing to realise is that the code from the class **`Die`** can be *re-used* because this class provides all of the necessary methods for simulating the behaviour of the **`Die`** in this game. In fact, in order to write this program all that we need to do is to encode the relatively simple game logic in another class called **`DieGame1`**. To assist you in this task a pseudo-code solution is provided on the next page and below there a few tips relating to the implementation of this design.

According to the design the first task is to create a **`Die`** object and initialise certain variables. This is achieved with the following sequence of Java statements:

```
Die aDie = new Die(6);
int score = 6;
boolean gameOver = false;
```

The first statement in the sequence accomplishes the following tasks

- an object, named **`aDie`**, instatntiating the class **`Die`** is created
- the face value is set to a random value in the closed interval [1,6]

Notice that to create a particular object of the class **`Die`** we need to use the **`new`** operator in conjunction with the constructor for that class. The other two statements are simply assigning specific values to an integer and boolean variable respectively.

There is not much to say about that part of the program which displays a start message and column headings; it is merely a matter of writing the appropriate strings and integer values using the standard **`println`** method.

Pseudo-code Design for Die Game: Six or Lose

```
create a Die object
set score to 6
set gameOver to false

print start message

print table header

while (not gameOver)
{
      throw die
      if die face equals 6
            set gameOver to true
      else
            set score to score - 1

      if score equals 0
            set gameOver to true
      printdie face and score
}

if die face equals 6
      write win message, score
else
      write lose message
```

Exercise 4: Creating and Controlling Multiple Objects

It is now time to look at how to develop a program which uses more than one die. To do this we will have to find some way of declaring and creating multiple objects of a single class. Fortunately, it is extremely easy to do this in Java. To declare and create two Die objects simply write:

```
Die die1 = new Die(6);
Die die2 = new Die(6);
```

The effect of these two statements is to create and initialise two distinct objects belonging to the class Die. We express this by saying that `die1` and `die2` are *instances* of the same class. This means that we can perform all of the **`Die`** operations, thus far defined, on either of these objects.

Note also that each object has its own distinct **`Die`** face value: the face value of **`die1`** is independent of the face value of the other Die. If this were not the case then the two Die would always show the same value which is not at all what we want.

Let us now consider how to use the two Die objects to create a somewhat more complicated dice game. Consider the description below:

> *This is a single player game. The player starts with a score of 0. The pair of dice are then thrown three times. For every double thrown, the player scores 1 point. At the end of the game the total score is displayed.*

As in the previous example our task is to write a program that will simulate this game. Some screen output for this program is presented below. In the game below no doubles are obtained so the player's score remains at zero.

```
Starting Doubles ...
die1        die2
*****       *****
2           4
3           2
1           3

Your score = 0
```

Notice that the player's score is only displayed at the *end* of the game.

In the next game the player obtains one double and so has a score equal to one.

```
Starting Doubles ...
die1       die2
*****      *****
3          3
6          1
5          2

Your score = 1
```

In the final example the lucky player throws three doubles and so obtains a score of three.

```
Starting Doubles ...
die1       die2
*****      *****
3          3
6          6
2          2

Your score = 3
```

Your task is to develop and encode the game logic in a new class called **DieGame2**. Some tips are provided below:

- use either `printf` of `String.format()` to manage two column layout
- use a `for` loop to control the throw of the two Die
- each iteration will involve throwing two die and displaying the resulting values
- use `System.out.print` if you don't want to break the line display

Implementation Note and a Challenge

For more complex games it may be worth implementing the game playing logic by means of a separate class. However, for games as simple as the two we are considering this is probably overkill although object oriented purists are welcome to adopt this design. If you do follow this kind of approach you might end up with something like this:

```
public class GameTester
{
      public static void main(String[] args)
      {
            DieGame1 game = new DieGame1();
            game.play();
      }
}
```

Of course this will involve modifying `DieGame1` (or `DieGame2`) and providing them with suitable constructors to initialise the game state and a `play` method. You are welcome to try!

Exercise 5. Building a Two-Player Game

Having developed a slightly more interesting game involving two dice objects our task is to consider how to produce a two-player rather than a single player game. When the game runs the screen output will look like this:

```
Dice1     Dice2
*****     *****
2         4
3         2
1         3

Player 1: score = 0

Dice1     Dice2
*****     *****
5         3
2         2
6         4

Player 2: score = 1

Player 2 wins!
```

How are we to write this program? Well, as we have seen from the previous examples we can use the already existing program which implements the class **`Dice`**. However, we now have a new factor to consider, namely the existence of two players. This should make us think that it might be useful to have a class to represent player objects. Now, for the purposes of this game a player is simply someone who has a certain score. This score can change over time and will also need to be displayed on screen. If we are to develop a player class we need to ask what methods it should provide, as well as think about the attributes of the objects of that class. Let us tackle the question of the object's attributes first. In the context of this game all that we need to know about a player, is the score which they have obtained.

On the next two pages I discuss the development of the **`Player`** class. The discussion is intended to revise some basic concepts involved in object oriented programming and is worth careful study, especially if you feel bit hazy about the fundamentals of this style of programming.

After developing the **`Player`** class you should now be in a position to implement a two-player version of our simple game involving a pair of dice.

We start as always with a constructor and instance variables (in this case there is only one).

```
public class Player

{
     public Player(int initialScore)
     {
          score = initialScore;
     }

     int score;
}
```

Notice that we have provided, not only an integer variable to represent the score attribute of a player, but at the same time have defined a constructor method for this class. You may have wondered why this constructor method takes a parameter rather than simply sets the value of **`score`** to zero. In other words why didn't we just write

```
public class Player
{
     public Player()
     {
          score = 0;
     }

     int score;
}
```

Well the answer to that question is that by providing a parameter to this constructor we are making the **`player`** class a little more *re-usable*. Thus later we may want to develop a different two player game where the initial score is not equal to zero. By providing the constructor **`Player`** with a parameter we can *re-use* this class to develop other kinds of dice game. For example, some dice games may start by awarding players a score greater than zero. As you will know, one of the goals of object oriented programming is to produce units of software, in particular classes, which can be used over and over again to solve a whole variety of programming problems. For our new class **`Player`** we provide a pair of simple *mutator* and *access* methods. These are listed below:

```
// Mutator method to increases player's score

public void setScore(int points)
{
     score += points;
}


// Access method to return value of player's score
```

```
public int getScore()
{
     return score;
}
```

Notice that the method set**`Score`** uses the combined assignment and arithmetic operator **`+=`** to increment the player's score. The same effect, of course, could have been obtained using the statement below

```
     score = score + points;
```

Notice also that because we have parameterised this method we have made the **`Player`** class a little more flexible so that it could handle games where the score increases by more than one point. We can now put all of this together to produce an implementation of the class **`Player`**. The listing for this class is provided below.

```
// Defines methods and attributes for Player class

public class Player
{
     // Player constructor

     public Player(int initialScore)
     {
          score = initialScore;
     }

     // Increases score

     public void setScore(int points)
     {
          score += points;
     }

     // Returns value of players score

     public int getScore()
     {
          return score;
     }

     // Instance variable to store Player's score

     private int score;
}
```

Assessed Laboratory Work

PIg is a well known two-player, jeopardy style game, widely used in computing and mathematics courses to teach concepts of probability. The rules for the game of *Pig* are given below:

Players

Best with two players, but can work with more. (But note that the downtime between your turns grows longer with each additional player.)

Equipment

The game is played with a single six-sided die.

Goal

Be the first player to reach 100 points.

Game Play

On each turn, a player rolls the die repeatedly until either:

- A one is rolled
- The player chooses to hold (stop rolling)

If a one is rolled, that player's turn ends and no points are earned.

If the player chooses to hold, all of the points rolled during that turn are added to his or her score.

Scoring Examples

Example 1: Sherri rolls a 3 and decides to continue. She then chooses to roll seven more times (6, 6, 6, 4, 5, 6, 1). Because she rolled a 1, Sherri's turn ends and she earns 0 points.

Example 2: Craig rolls a 6 and decides to continue. He then chooses to roll four more times (3, 4, 2, 6) and decides to hold. Craig earns 21 points for this turn (6+3+4+2+6=21).

Game End

When a player reaches a total of one hundred or more points, the game ends and that player is the winner.

Your task is to code this game. A simple way of doing it is for the program to alternately invite two human players to take their turn and to keep rolling until a one is thrown or the player decides to stop rolling and pass the dice to the other player. A more complex and challenging approach might be to automate the game entirely and provide each virtual player with their own game playing algorithm.

Apart from the classes **`Die`** and **`Player`** you are free to design the program as you wish.

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:

• Implementation of the class `Die` and `DieTester`	20%
• Implementation of the class `Player` and `PlayerTester`	20%
• Implementation of the Pig game	
○ Design	20%
○ Correctness	30%
• Comments, layout and coding conventions	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.