9 - CLASSES

9.1 Introduction

Classes in Java are a form of *encapsulation,* where data and methods can be gathered together to serve a common purpose. For example, the `String` class contains methods such as `charAt()`, `substring()`, `length()` that operate on strings. The `RandomAccessFile` class contains methods such as `setLength()`, `seek()`, `close()` that operate on files. The classes that we have used so far in this course have been predefined by other programmers:

- `Math`
- `Stdin`
- `Stdout`
- `Console`
- `RandomAccessFile`
- `String`

In these examples we have the two Java class types: *non-instantiating* and *instantiating*. `Math`, `Stdin`, and `Stdout` are examples of non-instantiating classes. The others are instantiating. We now look at these in more detail.

9.2 Non-Instantiating Classes

An non-instantiating class is one that gives access to its data and methods *without instantiating* an object. For example, the statement

```
double area = Math.pow(23, 2);
```

lets us raise a number to an exponent by writing the class name `Math`, followed by a period, followed by the method `pow()`. Note that no object has been instantiated in this case.

We also have access to the internal data of the `Math` class. In the statement

```
double circumference = 2 * Math.PI * 8;
```

the constant `PI` has been predefined inside the `Math` class and we have access to its value by preceding the constant `PI` with the name of the class. The `Math` class has two constant values: `PI` and `E` and over 50 methods. A description of the `Math` class can be found at

http://docs.oracle.com/javase/7/docs/api/java/lang/Math.html

Some more examples of statements that make use of this type of class:

```
String name = Stdin.readLine();
Stdout.println("School will be over soon");
double alpha = Math.sin(2.3);
```

Defining our own non-instantiating class

All the classes we have created in this course so far contain a `main()` method and possibly other methods. We now learn to define a class without a `main()` method.

Non-instantiating classes may contain constants, variables and/or methods. Each one of these is optional, that is, a class may contain only methods, or only constants, or only variables, or any combination of these. All constants, variables and methods are defined `static` since the *qualifier* `static` makes these variables and methods not associated with any particular object. A `static` constant/variable/method is available to other classes at any time while the program is running. Non-instantiating classes are defined in their own separate file with the very same filename as the class's name.

The internal structure of the class is,

```
<import statements>
public class <class name> {
    <static constants>
    <static variables>
    <static methods>
}
```

The following is an example of a definition of a non-instantiating class:

```
import java.io.*;
public class FileOperations {
    public static RandomAccessFile createFile (String fileName) {
        RandomAccessFile f = null;
        try {
            f = new RandomAccessFile (fileName, "rw");
            f.setLength(0);
        }
        catch (IOException e) {
            System.out.println("***error creating file" + fileName);
        }
        return f;
    }

    public static String readFromFile(RandomAccessFile f) {
        String s = "";
        try {
            s = f.readLine();
        }
        catch(IOException e) {
            System.out.println("***error reading from file");
        }
        return s;
    }

    public static void writeToFile(RandomAccessFile f, String data) {
        try {
            f.writeBytes(data);
        }
        catch(IOException e) {
            System.out.println("***error writing to file");
        }
    }

    public static void closeFile(RandomAccessFile f) {
        try {
            f.close();
         }
         catch(IOException e) {
            System.out.println("*error closing file");
         }
    }
}
```

In a separate class we then place the `main()` method and the necessary calls are made to the methods of the class above:

```
import java.io.*;
public class Filing {
    public static void main(String[] args) {
        RandomAccessFile f = FileOperations.createFile("contacts.txt");
        if (f != null) {
            FileOperations.writeToFile(f, "P Jones 416-353-2281\r\n");
            FileOperations.writeToFile(f, "S Wore 647-223-4844\r\n");
            FileOperations.writeToFile(f, "C Chow 905-345-9888\r\n");
        }
    }
}
```

The method calls are preceded by the class. The method `writeToFile()` needs to be qualified by the name of the class `FileOperations`

9.2 Exercises

1. Use the following link to find the class `Integer` (class in `java.lang`)

   http://docs.oracle.com/javase/7/docs/api/index-files/index-1.html

   Choose any two constants (fields), read their definitions, and write a short description of what the definitions mean.

   Choose any three static methods that return a primitive type, read their definitions, and write a short description of what the methods do.

2. Write a non-instantiating class with the header

   `public class Statistics`

   and include in this class any two of the following:

   - A method that returns the mean
   - A method that returns the mode
   - A method that returns the median
   - A method that returns the maximum
   - A method that returns the minimum

   Each method takes in four integers as parameters.

   Then create a separate class containing the `main()` method and the calls to the two methods that you included in the class `Statistics`

9.3 Instantiating Classes and Object Oriented Programming

The tools that we have learned so far are necessary for building any program in Java, or any other language for that matter. The type of programming we have been using is known as *procedural programming*. Languages such as standard Pascal, C, Turing, or PL/1 are of this type. In these languages we, as programmers, write programs as sequences of commands for the computer to execute. The introduction of *objects* into programming languages creates a shift in the way we approach programming. Objects are defined much like variables, but objects, unlike variables, have *properties* and *behaviours.*

In object oriented languages such as Java, C++, Objective C, C#, Object Oriented Turing, these properties and behaviours are defined by emulating real life objects. For example: We can define an object such as a tennis ball. Then we give the object properties such as colour, size and speed; we can then define its behaviour such as its response to bouncing off the floor. Many times the behaviour of the ball can affect its properties.

In object oriented programming a sort of *blue print* is first created that contains the definition of the object. This is done through the use of classes. Properties and behaviours are all defined in the class. To create an actual object, we must *declare* it like a regular variable, but we must also *instantiate it.* The instantiation process is the one responsible for the actual creation of the object. We can instantiate as many objects as we want using the same class, and Java will maintain these as separate objects.

There exist many predefined classes in Java that allow us to instantiate objects and use them. We will start with these. Then, we can create our own classes which will allow us to instantiate objects that we ourselves have defined.

Here is the internal structure of a class:

```
public class <ClassName> {
     <variables>
     <constructors>
     <methods>
}
```

Here is an example of a class that we will call `Customer`*.* An explanation follows after:

```
public class Customer {
     // variables
     private String name = "";
     private int age = 0;
     private double balance = 0.0;

     // constructors
     public Customer() {
          name = "";
          age = 0;
          balance = 0;
     }

     public Customer(String n, int a, double b) {
          name = n;
          age = a;
          balance = b;
     }

     // access methods
     public String getName() {
          return name;
     }

     public void setName(String n) {
          name = n;
     }

     public int getAge() {
          return age;
     }

     public void setAge(int a) {
          age = a;
     }

     public double getBalance() {
          return balance;
     }

     public void setBalance(double b) {
          balance = b;
     }

     // output method
     public String toString() {
          return "Customer name: " + name + "\n" +
                 "Age: " + age + "\n" +
                 "Balance: " + balance;
     }
}
```

The example above follows the essential structure:

```
     public class <ClassName> {
          <variables>
          <constructors>
          <methods>
     }
```

Let's look at these individually:

ClassName

The ClassName follows the same conventions as we have used before when we created classes that contained static methods. The name of the class starts with an upper case letter. If the class name has more than one word in it, we write each word starting with upper case, and join them together. For example:

```
     public class TennisBall {
          ...
     }
```

The class name has to be saved with a filename that is exactly like the class name. Operating systems such as Windows XP will not care about upper or lower case. Unix, Linux, MacOS-X and other Unix based systems will care. Since Java programs are designed many times with the internet in mind, we ultimately do not know on which machine our system will run. Our very classes may end up being part of a larger package of classes to be used by other developers. Better to be safe and follow conventions and thus avoid easily preventable problems.

Variables and Access Methods

Variables in a class determine the properties of an object. By having a class we can contain much more information than is available with a single variable. Within a class we can have as many variables as we want, some of which are primitives, some of which are objects. For example, our `Customer` class will allow us to maintain information about a customer on their name, age and balance. We can include other information such as address, telephone, e-mail, social insurance number, place of birth, marital status, education, present employment, and much more. We virtually have no limitation to the amount of information we can store.

In Java, variables within a class are declared `private`. This means that the variable is accessible only within the class. Because it is possible that we will need access to the variables from outside the class, we create two *access methods* for each variable: One to retrieve the value of the variable and the other to assign a value to the variable. In our `Customer` class the variable `age` is accessed *indirectly* by using these two methods:

```
public class Customer {
     ...

     public int getAge() {
          return age;
     }

     public void setAge(int a) {
          age = a;
     }

     ...
}
```

By using methods to indirectly access the internal variables of an object, we maintain *encapsulation*. Variables can be declared `public`, `private`, or `protected`. The guideline of Object Oriented Programming is to restrict as much as possible direct access to an object's variable. The top priority is to give only, and only an object, access to its own variables (*private)*. Then, when we learn about inheritance, where objects inherit the properties of other objects, we will declare some variables *protected*, thereby giving direct access to variables that have been inherited (More to come soon.) Declaring a variable *public* should be done only in the context of a main method or within the main class. This entire mechanism is in place in order to avoid potential ambiguity and maintain programs in independent modules.

Constructors

The constructor is a special method that is used to build the object that is being instantiated. A constructor has these special properties:

1. Its name is the same as the name of the class. It is the only method whose name starts with an upper case letter.

2. It does not have a return type in its declaration. In our *Customer* class, the constructor method is declared,

```
public Customer(<optional parameters>) {
     ...
}
```

3. If a constructor method is not included in the class, Java will automatically insert one with no parameters and no statements in its body. This implicitly inserted constructor will have the same effect as the following,

```
public Customer() {
}
```

4. The purpose of the constructor is to initialize the variables contained within the class. A constructor method may contain as much code as is necessary, including loops, decision statements, and calls to other methods. Parameters in the constructor definition are often needed so that the variables may be initialized to those parameters. When an object is being instantiated, the call to the constructor will contain the values passed to the parameters. For example, in our `main()` method we would have the following to instantiate an object of the class `Customer`*:*

```
Customer c = new Customer("John", 32, 12553.25);
Customer d = new Customer("Suzie", 18, 32444.76);
```

The effect of these two statements is that two objects, namely `c` and `d` are created with the values passed in the parameters. Each one of these two objects has access to its own set of variables and to all of the methods defined in the class. They are completely independent of each other.

5. A class may contain more than one constructor. This is possible in Java because of overloading. Therefore, these two constructor definitions are perfectly acceptable to the compiler,

```
public Customer() {
     name = "";
     age = 0;
     balance = 0;
}

public Customer(String n, int a, double b) {
     name = n;
     age = a;
     balance = b;
}
```

Circumstances do arise when an object may have to be created but we may not know the initial values. In that case a constructor without parameters may be necessary.

Methods

Methods within a class determine the behaviour of the object. They are also used to access variables indirectly, to print out the contents of the object, and to perform tasks, such as processing events like mouse clicks.

In our class above, we have methods that let us access the values of the variables and set values to the variables. Later on we will include other methods that define the behaviours. (See our program on fraction operations: Addition, subtraction, multiplication, division, and simplification.)

A special method within a class is the `toString()` method. Whenever we have wanted to print something on the screen we have used `Stdout.print(<item>)`. From now on, we will use the method `System.out.print(<object>)`. This new version allows us to pass an entire object to the printing method. Java will invoke the special method `toString()` that we define in our class and whatever string we return will be printed. Here is an example with our `Customer` class:

```
public String toString() {
     return "Name: " + name + "\n" +
            "Age: " + age + "\n" +
            "Bal: " + balance;
}
```

If we have the following code in our main method:

```
Customer c = new Customer("John", 32, 12553.25);
System.out.println(c);
```

The following output will be produced:

```
Name: John
Age: 32
Bal: 12553.25
```

9.3 Exercises

1. Create a class named `Fruit`. Then define/declare/initialize,

   a. A private string object that will contain the name of the fruit object
   b. A private string object that will contain the country of origin of the fruit
   c. A private double primitive that will contain the price
   d. A default constructor method that initializes all private variables
   e. A constructor method with signature

   ```
   public Fruit(String n, String c, double p)
   ```

   that initializes the private variables to the given parameters in the given order.

   f. Three void methods to set the values of the private variables
   g. Three typed methods to get the values of the private variables
   h. The `toString()` method.

   i. Create a class named `Fruits` and include a `main()` method. Within it create four fruit objects and fill them with data of your own. Then print out their contents.
   j. Using the appropriate access method, set the country of one of the fruits to "Philippines"
   k. Using the appropriate access method, set the price of one of the fruits to 1.35
   l. Print only the price of a fruit using its access method.

2. Create a class named `Rectangle`. Then define/declare/initialize,
   a. A private double primitive that will contain the length
   b. A private double primitive that will contain the width
   c. A default constructor method that initializes all private variables
   d. A constructor with signature

   ```
   public Rectangle(double l, double w)
   ```

   that initializes the private variables to the given parameters in the given order.

   e. Two void methods to set the values of the private variables
   f. Two double type methods to get the values of the private variables
   g. A double type method that returns the area of the rectangle
   h. A double type method that returns the perimeter of the rectangle
   i. A boolean type method that returns whether the rectangle is a square
   j. Create a separate class with a `main()` method and test the `Rectangle` class

3. Create a class named `Student`. Then define/declare/initialize,
   a. Private string *objects* that will contain:
      i. The student number: A nine digit number without any spaces.
      ii. The student name: This will always be a first name followed by a space followed by a last name.

   b. Private string objects that will contain course codes for two courses

These codes will have the pattern AAADDD, where A stands for any upper case letter and D stands for any number. For example: MAT101

c. Private double variables that will contain the numeric grade for each course. These numbers will be stored as a grade from 0 to 100.

d. The default constructor.

e. A constructor with the signature

```
public Student(String stNo,
               String stName,
               String courseCode1,
               String courseCode2,
               double mark1,
               double mark2) {

     < constructor code goes here >

}
```

f. The methods that set the values of the private variables. We need to write 6 methods since there are 6 variables. Here is the first one (assuming that the private variable for the student number is called `studentNumber`)

```
public void setStudentNumber(String stNo) {
     studentNumber = stNo;
}
```

g. The methods that get the values of the private variables. We need to write 6 methods since there are 6 variables. Here is the first one (assuming that the private variable for the student number is called `studentNumber`):

```
public String getStudentNumber() {
     return studentNumber;
}
```

h. The `toString()` method that will print out the contents of the student object. This method will return a nicely formatted string as follows:
   i. The student numbers will contain dashes between the 3rd and 4th digits and between the 6th and 7th digits.
   ii. The name will print only the initial of the first name, followed by a period, a space and then the last name
   iii. The course code will as is
   iv. The mark will print with an added percentage sign at the end.

   Here are some output samples:

```
330-332-123 J. Smith MAT101 86%
123-171-776 A. Cole ENG100 65%
644-271-112 F. Rusianki CSC201 76%
```

i. Create a separate class with a `main()` method and test the `Student` class

4. Create a class name `Fraction`. Then Then define/declare/initialize,
   a. A private integer `numerator` initialized to zero
   b. A private integer `denominator` initialized to zero
   c. A private boolean `undefined` initialized to `true`
   d. A default constructor that initializes the instance variables
   e. Another constructor with integer parameters `n` and `d` which are assigned to `numerator` and `denominator` respectively. Then the variable `undefined` is set to `denominator == 0`
   f. Include six access methods to set and get the values of all these variables
   g. Include a `toString()` method that will print the fraction object in readable format. For example, if `numerator` is `5` and `denominator` is `7` the `toString()` method returns `5/7`. If both numerator and denominator are negative, the `toString()` method returns a positive fraction. If the denominator is `1`, the `toString()` method returns just `numerator`. If the fraction is undefined, the `toString()` method returns the word `undefined`
   h. Create a `main()` method in a separate class that tests your `Fraction` class. Instantiate four fractions and print each `Fraction` object with,

a. `numerator = 7, denominator = 5`
b. `numerator = -3, denominator =-2`
c. `numerator = -8, denominator = 1`
d. `numerator  = 5, denominator = 0`

The output should be,

```
7/5
3/2
-8
undefined
```

i. Implement fraction addition, subtraction, multiplication, and division
j. Implement fraction simplification using the `euclid()` method below for the greatest common divisor of two positive integers `m` and `n`

```
private static int euclid(int m, int n) {
     while (m != 0 && n != 0) {
          if (m > n)
               m = m - n;
          else
               n = n - m;
     }

     if (m != 0)
          return m;
     else
          return n;
}
```

5. Create a class named `Linear` that maintains the equation of a line in exact form, i.e., uses fractions instead of approximations. In this class define,

   a. A private `Fraction` that will contain the slope of the line
   b. A private `Fraction` that will contain the y-intercept
   c. A `Fraction` type method that computes the y-coordinate from a given x-coordinate (Assume that the x-coordinate is also a `Fraction`)

d. A `Fraction` type method that computes the x-coordinate from a given y-coordinate (Assume that the y-coordinate is also a `Fraction`)
e. The `toString()` method that prints out the equation of the line in exact symbolic form. For example:

```
y = 3/5(x) + 7/11
```