JAVA CLASSES

A class is a code that does a particular job. A class is the code behind an object. When we reference to the class, that is when we put the class into action, it is called an object.

We write a class to represent a concept or something. This concept or thing would possess properties and methods. Properties are the features that the object's class has. Methods are actions that the object (or class) can perform. It is good to write a class to abstract a particular concept or object, if not it would lead to redundancies.

As an example, we'll write a class that deals with exams. We'll make things simple. We'll record the name of the person taking the exam which exam it was, the score out of 50, and a grade.

To start, create a new Java project, call the package 'exams' and then change the name of the Main method to "ExamDetails." This is the code version:

```
package exams;
public class ExamDetails {
      public static void main (String []args) {
      }
}
```

We'll create a second class to handle the exam data. So, in NetBeans, click File from the menu bar at the top. From the file menu, select New File. Highlight Java from the categories list and Java Class from the file types list. Then click Next. On the next screen, enter StudentResults as the class name. Then click Finish. NetBeans will create a second class in your project.

Variable Scope: Local and Field Variables

Variables set up inside of methods are only available to those methods. They are not available to other methods. They are said to have *local scope*.

However, you can set up variables outside of methods that all the methods in your class can see. These are known as *Field variables* (or *Instance*

variables). You set them up in exactly the same ways as any other variable. For example:

```
package exams;
public class StudentResults {
      String Exam_Name;
      String Exam_Score;
}
```

We've set up two string variables. As the names of the fields suggest, the string will hold an Exam name and its score. These two fields will be available to all the methods that we write in this class, and won't be local to any one method. They are said to have *global scope*.

Now we can call the global or field variables in the main method of another Java class in the same project.

```
package exams;
public class ExamDetails {
  public static void main (String []args) {
    StudentResults aStudent= new
StudentRestults();
    aStudent.Exam_Name= “VB NET”;
    String exam= aStudent.Exam_Name;
    System.out.println (exam);
  }
}
```

The two fields that we set up are on the list. They are not methods, but something called a property. The fact that they are on the list at all means that they have global scope. If they had local scope they wouldn’t be on the list.

We’ve selected the Exam_Name field from the list and assigned it the value “VB NET.” The next line then gets the value of Exam_Name from the aStudent object. The result is stored in a variable called exam, and then printed out. When you run the program, the output is the string “VB NET.” In essence, the two variables we’ve set up are available to both classes, i.e. they have global scope.

Notwithstanding, it's not a good idea to make field variables global like this. You may lose track of just what values are being held in them, and it therefore makes debugging your code a lot harder. It's considered good coding to narrow the scope of variables. To make a field variable available only to a particular class, you add the Java keyword private just before the class declaration, like this:

```
package exams;
public class StudentResults {
   private String Exam_Name;
   private String Exam_Score;
}
```