

Data Classes and Object-Oriented Programming

Data classes can be motivated by the need to create data structures that have grouped together a number of variables of simpler type (ints, Strings, arrays) in order to represent the structure of a complex real-world object. Object-oriented languages go further in associating with the data structures the operations that can represent what behaviors we want to model. Here is a quote from the Sun Java tutorial:

“Real-world objects share two characteristics: They all have *state* and *behavior*. Dogs have state (name, color, breed, hungry) and behavior (barking, fetching, wagging tail). Bicycles also have state (current gear, current pedal cadence, current speed) and behavior (changing gear, changing pedal cadence, applying brakes). Identifying the state and behavior for real-world objects is a great way to begin thinking in terms of object-oriented programming.”

In Java, the data class is used to provide the definition of how to represent the state of an object in terms of a group of variables and to provide methods that define how the behaviors of the object will occur.

Each class can be considered as a template and that objects are created by creating an “instance” of the template. Each instance can have the data that represents one object of the class.

Note that there are classes not used as data classes, but primarily for computation. For example, a class may have a main method that is used to start the computation. It is expected that there will only be one instance of this method and it is declared as *static*.

Class Definitions

The class definition is not a program by itself. It can be used by other programs in order to create objects and use them.

Diagram of a class definition:

```
public class <classname> {  
    <fields>  
    <constructor functions>  
    <methods>  
}
```

The fields are declarations of variables that represent the class. These variables can have any valid type.

The constructor function is used to create an instance of the class, and may also initialize the field variables. Calling this function allocates the instance.

The other methods can operate on the fields in order to implement whatever operations need to be done with this class.

Here is one implementation of an example representing the employees of a company:

```
class Employee {  
  
    // fields  
    String empName;  
    double salary;  
    String hireDate;  
  
    // constructor function creates and instance and initializes fields  
    public Employee (String startName, double startSalary,  
                     String startHireDate){  
        empName = startName ;  
        salary = startSalary;  
        hireDate = startHireDate;  
    }  
  
    // other methods to provide functionality  
    void raiseSalary(double rate) {  
        salary = salary + (salary * rate);  
    }  
  
    double getSalary() {  
        return salary;  
    }  
  
    String getHireDate() {  
        return hireDate;  
    }  
  
    void setEmpName(String newName) {  
        empName = newName;  
    }  
  
    void printEmployee() {  
        System.out.println("Employee Name:"+empName+  
                            " salary:"+salary+" hire date:"+hireDate);  
    }  
}
```

Note that the methods of the class just refer directly to the field variables as they would for any other variables in any type of class.

Using a Class

Since a class definition gives a template of a number of fields and methods, a program that wants to use objects of the class would create one or more instances of the class.

To create an instance, declare a variable that has the <classname> as its type, and uses the “new” keyword with the constructor method and any parameters that the constructor method needs to do initialization.

```
<ClassName> <varname> = new <ClassName> (<param1>, . . . , <paramN>)
```

Example:

```
Employee emp1 = new Employee ( “Carl Worker”, 35000.0, “Jan2005”);
```

This statement will allocate memory for an instance of the Employee class that will contain a memory allocation for each of the field variables and initialize the employee name to “Carl Worker”, the salary to 35000.0, and the hire date to “Jan2005”.

Once a class instance has been created in a variable, then the dot notation is used to use the fields and methods in the class.

```
<varname> . <fieldname>  
<varname> . <methodname>
```

Here is an example program that creates two instances of the Employee class and uses the fields and methods.

```
class EmployeeDemo {  
    public static void main(String[] args) {  
  
        // Create two different Employee objects  
        Employee emp1 = new Employee ( “Carl Worker”, 35000.0, “Jan2005”);  
        Employee emp2 = new Employee ( “Barbara Boss”, 70000.0, “Jan1998”);  
  
        // Invoke methods on those objects  
        emp1.raiseSalary( .10);  
        emp1.printEmployee();  
  
        emp2.raiseSalary( .03);;  
        emp2.setEmpName ( “Barbara Harris”);  
        emp2.printEmployee();  
    }  
}
```

Other examples of using classes

In Java, all of the built-in programs and data come from classes and are described in the Java API (Application Programming Interface).

Some examples that we have used so far include the GUI components that we use to make a form design. But in these cases, NetBeans helps by generating some of the code for us. So when we drag components onto the form, NetBeans generates code that creates instances of the classes representing those components.

```
buttonGroup1 = new javax.swing.ButtonGroup();
jLabel1 = new javax.swing.JLabel();
jLabel2 = new javax.swing.JLabel();
jButton1 = new javax.swing.JButton();
jRadioButton1 = new javax.swing.JRadioButton();
jRadioButton2 = new javax.swing.JRadioButton();
jRadioButton3 = new javax.swing.JRadioButton();
jTextField1 = new javax.swing.JTextField();
jTextField2 = new javax.swing.JTextField();
```

NetBeans also automatically generates code to change the size and appearance of the components and other things like:

```
buttonGroup1.add(jRadioButton1);
```

Then when we write the code for the button `actionPerformed` method, we can also use methods for the components. For example, we can get and set the text appearing in a label or textfield:

```
String perName = jTextField1.getText();
jLabel1.setText ("Name " + perName);
```

There are some methods that can be used without creating an instance of the class. These methods are declared to be static methods, and they don't use any one instance of the class. For example, when we convert a `String` to an `int`, the method `parseInt` is *static* and is used just by giving the name of the class, `Integer`:

```
Integer.parseInt(String x)
```

And the `Math` class provides a number of mathematical functions as static methods, for example, `Math.pow(double a, double b)` // computes a raised to the power b

Data Encapsulation

When designing a program, it is important to design a class for each type of data object and to use methods to code every operation on objects of the class. This accomplishes two important principles:

Modularity: Conceptually all the details of using the class are defined by what methods to call, making each piece of the program separated into its own class.

Data Hiding: Other parts of the program should not use the fields directly; only use them through the methods. This means that the class can be given a different implementation and the programs using it will not have to be changed.

As a result of these principles, it is recommended that most fields of a class be made private, unless it really is a field that should be publicly accessible. This means that these fields cannot be accessed directly from another class using the dot notation. For example, in the Employee class:

```
// fields
private String empName;
private double salary;
private String hireDate;
```

Other programs can no longer use “emp1.salary”, for example. If other programs need to get the value of the field, an accessor get method is given. An accessor method already given is

```
public double getSalary() {
    return salary;
}
```

And if other programs need to assign the value of the field, a set method is given. For example,

```
public void setEmpName(String newName) {
    empName = newName;
}
```