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.

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 the bicycle example adapted from the Sun Java tutorial:

```
class Bicycle {

       // fields
       int cadence = 0;
       int speed = 0;
       int gear = 1;
     // constructor function creates and instance and initializes fields
      public Bicycle (int startcadence, int startspeed, int startgear){
         cadence = startcadence;
         speed = startspeed;
         gear = startgear;
       }

      // other methods to provide functionality
       void changeCadence(int newValue) {
            cadence = newValue;
       }

       void changeGear(int newValue) {
            gear = newValue;
       }

       void speedUp(int increment) {
            speed = speed + increment;
       }

       void applyBrakes(int decrement) {
            speed = speed - decrement;
       }

       void printStates() {
            System.out.println("cadence:"+cadence+
                        " speed:"+speed+" gear:"+gear);
       }
}
```

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:

Bicycle bike = new Bicycle (30, 1, 10);

This statement will allocate memory for an instance of the Bicycle class that will contain a memory allocation for each of the field variables and initialize the cadence to 30, the gear to 1, and the speed to 10.

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 Bicycle class and uses the fields and methods.

```
class BicycleDemo {
     public static void main(String[] args) {

          // Create two different Bicycle objects
          Bicycle bike1 = new Bicycle(30, 1, 10);
          Bicycle bike2 = new Bicycle(10, 1, 5);

          // Invoke methods on those objects
          bike1.changeCadence(50);
          bike1.speedUp(10);
          bike1.changeGear(2);
          bike1.printStates();

          bike2.changeCadence(50);
          bike2.speedUp(10);
          bike2.changeGear(2);
          bike2.changeCadence(40);
          bike2.speedUp(10);
          bike2.changeGear(3);
          bike2.printStates();
     }
```

```
}
```

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 bike class:

```
// fields
private int cadence = 0;
private int speed = 0;
private int gear = 1;
```

Other programs can no longer use "bike1.cadence", for example. If other programs need to get the value of the field, an accessor get method is given. For example,

```
public int  getCadence() {
       return cadence;
}
```

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

```
public void  setCadence(int newcadence) {
       cadence = newcadence;
}
```