CREATING YOUR OWN CLASSES IN JAVA

Up to now, the programs we have created were comprised of just one class. Now it's time to learn how to create programs that have more than one class.

DECLARING A CLASS

The first thing you will need to do when creating your own class is declare the class. The class declaration must include the access level, the keyword **class**, and the class name. The body of the class contains **fields**, **constructors**, and **methods**.

Fields define the public or private fields of the class.

Methods define the methods of the class.

Constructors are blocks of code that are similar to methods but are run to initialize an object when an instance is created. A constructor must have the same name as the class itself and, unlike methods, does not have a return type.

A class will, therefore, take the following format:

```
<access level> class <class name> {

   <fields>
   <constructors>
   <methods>
}
```

Just as there are rules for naming variables in Java, there are also rules for naming classes:

1. Begin the class name with a capital letter
2. Use nouns for your class names
3. Avoid using the name of a Java API class

Here's a simple example of a class file called **Dice** which includes a method called **roll()**:

```
import java.util.Random;

public class Dice {

   private Random rnd;

   public Dice() {

      rnd = new Random();
   }
```

```
   public int roll() {

      int randomRoll = rnd.nextInt(6) + 1;
      return randomRoll;
   }
}
```

If we wanted to use this class in a program called **DiceProgram**, our program would have to look something like this:

```
public class DiceProgram {

   public static void main(String[] args) {

      // Declare and initialize Dice object
      Dice d = new Dice();

      // Generate and output a roll
      System.out.println("You rolled a " + d.roll () + "!");
   }
}
```

FIELDS

A **field** is a variable that is defined in the body of a class and outside any of the methods within the class. Fields, which are also called class variables, are available to all the methods of a class. If the field specifies the **public** keyword, the field or variable is visible outside of the class. The word **public**, in other words, means that there are no restrictions on how these instance variables are used. If you don't want the field to be visible outside of the class, use the **private** keyword instead.

The following are examples of public fields:

```
public int xPos;
public double hourlyWage = 8.25;
public String name;
```

Private fields are declared in the same way, the only difference is that you use the word private instead of public:

```
private int xPos;
private double hourlyWage = 8.25;
private String name;
```

Although fields can be labeled either **public** or **private**, it is considered good programming practice to make all instance variables **private** and most methods **public**. Normally, a method is private only if it is being used solely as a helping method.

CONSTRUCTORS

A **constructor** is a block of code that's called when an instance of an object is created. A constructor is very similar to a method, but with a couple of key differences:

- A constructor doesn't include a return type
- The name of the constructor must be the same as the name of the class
- Unlike methods, constructors are not considered to be members of a class
- A constructor is called when a new instance of an object is created

The most common reason for creating a constructor is to provide initial values for class fields when you create the object. For example, let's say you have a class named **Dice** that sets the number of dice the user wants to use by taking an integer value from the user. You can create a constructor for the **Dice** class as follows:

```
public class Dice {

   private int numDice;
   private int possibleValues;
   private int max;

   public Dice(int x) {

      numDice = x;
      max = numDice * 6;
      possibleValues = max – numDice + 1;
   }

   public Dice() {

      numDice = 1;
      max = 6
      possibleValues = 6;
   }
}
```

The second **Dice** constructor is a default constructor that sets **numDice** to 1 by default, sets the **max** variable to 6, and sets the **possibleValues** variable to 6.

Our **roll()** method will, of course, need to be modified so that it can roll the number of dice that the user wants to use and outputs the result of the roll:

```
public int roll() {

   int randomRoll = rnd.nextInt(6) + 1 * possibleValues +
      numDice);
   return randomRoll;
}
```

Now when we create a **Dice** object in our **DiceProgram** application, we can specify the number of dice we want to use as follows:

```
public class DiceProgram {

   public static void main(String[] args) {

      // Declare and initialize Dice object
      Dice d = new Dice(2);

      // Generate and output 10 rolls
      for (int i = 1; i <= 10; i++)
      {
         System.out.println("You rolled a " + d.roll() + "!");
      }
   }
}
```

It is common for constructors to take no arguments. Such a constructor is called a **no-arg** or **no-argument** constructor. If you define a class and include no constructors of any kind, then a no-argument constructor is automatically created. This no-argument constructor doesn't do much, but it does give you an object of the class type. So, if the definition of the class **Dice** contains no constructor definitions, then the following is legal:

```
Dice d = new Dice();
```

If, however, your class definition includes one or more constructors, then no constructor is generated automatically. So, for example, if you include one or more constructor that takes one or more arguments, but you do not include a no-argument constructor in your class definition, then there is not a no-argument constructor and the following is illegal:

```
Dice d = new Dice();
```

To avoid such problems, you should always include a no-argument constructor in any class you define. If you do not want the no-argument constructor to initialize any instance variables, you can simply give it an empty body when you implement it. A no-argument constructor that does nothing but create an object would look something like this:

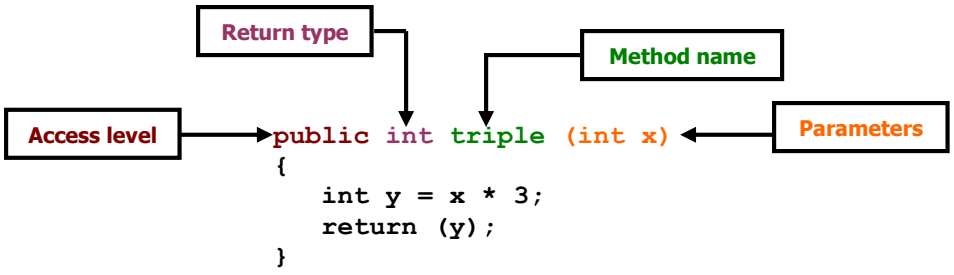
```
public class Dice {

   public Dice() {

      // Do nothing
   }
}
```

METHODS

A method declaration includes a header section and a body section. The header includes the following sections:



The **access level** determines whether or not other classes can call the method. The access level can either be **public**, in which case any method can access the method, or **private**, in which case only methods in the same class are allowed to access it.

The **return type** specifies what kind of data (e.g. int, double, String, boolean, etc.) is to be returned back after this method has completed its task(s). If the method does not need to return a value, the return type **void** is used.

Method name specifies the name of the method. Method names must adhere to the same naming conventions as variable names.

Parameters represent the data that is sent to the method so that the data can be processed inside of the method when it is invoked. Sometimes a method may not need values to perform its task, in which case the brackets remain empty.

ACCESSOR AND MUTATOR METHODS

Oftentimes you may need to do something with the data in a class object. **Accessor** and **mutator** methods allow users to access and change such data.

Accessor methods are often used to obtain data in a class object. For example, the methods getMonth(), getYear(), and getDay() are accessor methods that return the values of each instance variable. It is considered good programming practice to name accessor methods starting with the word **get**.

Mutator methods are used to change data in a class object. Generally, methods whose names begin with the word **set** are mutator methods.

OVERLOADING METHODS

A Java class can contain two or more methods with the same name as long as those methods accept different parameters. You are already familiar with some classes that include overloaded methods. For example, the **JButton** class includes two **setSize()** methods: one that required two integer values (representing the width and height of the button) and a second one that required a Dimension object:

```
setSize(int width, int height)
setSize(Dimension d)
```

The name of a method and the list of parameter types in the heading of the method definition is called the **method signature**. When you overload a method name, each of the method definitions in the class must have a different signature.