Classes and Objects

(Part 1, Classes)

Reading:

- Java Tutorial: Section *Classes and Objects* under section *Learning the Java Language*
- Textbook: Chapter 4 and 5
Classes:

public class Bicycle {

    // the Bicycle class has three fields
    public int cadence;
    public int gear;
    public int speed;

    // the Bicycle class has one constructor
    public Bicycle(int startCadence, int startSpeed, int startGear) {
        gear = startGear;
        cadence = startCadence;
        speed = startSpeed;
    }

    // the Bicycle class has four methods
    public void setCadence(int newValue) {
        cadence = newValue;
    }

    public void setGear(int newValue) {
        gear = newValue;
    }

    public void applyBrake(int decrement) {
        speed -= decrement;
    }

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

A class declaration for a MountainBike class that is a subclass of Bicycle might look like this:

```java
public class MountainBike extends Bicycle {

    // the MountainBike subclass has one field
    public int seatHeight;

    // the MountainBike subclass has one constructor
    public MountainBike(int startHeight, int startCadence, int startSpeed, int startGear) {
        super(startCadence, startSpeed, startGear);
        seatHeight = startHeight;
    }

    // the MountainBike subclass has one method
    public void setHeight(int newValue) {
        seatHeight = newValue;
    }

}
```

MountainBike inherits all the fields and methods of Bicycle and adds the field seatHeight and a method to set it (mountain bikes have seats that can be moved up and down as the terrain demands).

Declaring Classes
You've seen classes defined in the following way:

class MyClass {
  // field, constructor, and
  // method declarations
}

The class body (the area between the braces) contains all the code that provides for the life cycle of the objects created from the class:

- Constructors for initializing new objects
- Declarations for the fields that provide the state of the class and its objects
- Methods to implement the behavior of the class and its objects.
You can provide more information about the class, such as the name of its superclass, whether it implements any interfaces, and so on, at the start of the class declaration. For example,

class MyClass extends MySuperClass implements YourInterface
{
    // field, constructor, and
    // method declarations
}

Meaning: MyClass is a subclass of MySuperClass and that it implements the YourInterface interface.
The modifiers public and private, which determine what other classes can access MyClass, are discussed later in this lesson. The lesson on interfaces and inheritance will explain how and why you would use the extends and implements keywords in a class declaration. For the moment you do not need to worry about these extra complications.

In general, class declarations can include these components, in order:

- **Modifiers** such as public, private, and a number of others that you will encounter later.
- The class name, with the initial letter capitalized by convention.
- The name of the class's parent (superclass), if any, preceded by the keyword extends. A class can only extend (subclass) one parent.
- A comma-separated list of interfaces implemented by the class, if any, preceded by the keyword implements. A class can implement more than one interface.
- The class body, surrounded by braces, {}.
Declaring Member Variables
There are several kinds of variables:

- Member variables in a class—these are called fields
- Variables in a method or block of code—these are called local variables
- Variables in method declarations—these are called parameters

The Bicycle class uses the following lines of code to define its fields:

```java
public int cadence;
publish int gear;
publish int speed;
```

Field declarations are composed of three components, in order:

- Zero or more modifiers, such as public or private.
- The field's type.
- The field's name.

The public keyword identifies these fields as public members, accessible by any object that can access the class.
Access Modifiers

The first (left-most) modifier used lets you control what other classes have access to a member field. For the moment, consider only **public** and **private**. Other access modifiers will be discussed later.

- **public** modifier— the field is accessible from all classes.
- **private** modifier— the field is accessible only within its own class.
In the spirit of encapsulation, it is common to make fields private. This means that they can only be directly accessed from the Bicycle class. We still need access to these values, however. This can be done indirectly by adding public methods that obtain the field values for us:

```java
public class Bicycle {
    private int cadence;
    private int gear;
    private int speed;

    public Bicycle(int startCadence, int startSpeed, int startGear) {
        gear = startGear;
        cadence = startCadence;
        speed = startSpeed;
    }

    public int getCadence() {
        return cadence;
    }

    public void setCadence(int newValue) {
        cadence = newValue;
    }

    public int getGear() {
        return gear;
    }

    public void setGear(int newValue) {
        gear = newValue;
    }

    public int getSpeed() {
        return speed;
    }

    public void applyBrake(int decrement) {
        speed -= decrement;
    }

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

All variables must have a type. You can use primitive types such as int, float, boolean, etc. Or you can use reference types, such as strings, arrays, or objects.

Variable Names

Follow the same naming rules and conventions that were covered in the Language Basics lesson, Variables—Naming.

Be aware that the same naming rules and conventions are used for method and class names, except that

- The first letter of a class name should be capitalized, and
- The first (or only) word in a method name should be a verb.
Defining Methods

Here is an example of a typical method declaration:

```java
public double calculateAnswer(double wingSpan, int numberOfEngines,
                               double length, double grossTons) {
    //do the calculation here
}
```

The only required elements of a method declaration are the method's return type, name, a pair of parentheses, ( ), and a body between braces, { }. 

More generally, method declarations have six components, in order:

1. **Modifiers**—such as `public`, `private`, and others (you will learn about later)
2. The **return type**—the data type of the value returned by the method, or `void` if the method does not return a value
3. The **method name**—the rules for field names apply to method names as well, but the convention is a little different.
4. The **parameter list** in parenthesis—a comma-delimited list of input parameters, preceded by their data types, enclosed by parentheses, ( ). If there are no parameters, you must use empty parentheses.
5. An exception list—to be discussed later.
6. The method body, enclosed between braces—the method's code, including the declaration of local variables, goes here.

---

**Definition**: Two of the components of a method declaration comprise the *method signature*—the method's name and the parameter types.

**The signature of the method declared above is**: calculateAnswer(double, int, double, double)

---

**Naming a Method**

- By convention, method names should be a verb in lowercase or a multi-word name that begins with a verb in lowercase, followed by adjectives, nouns, etc
- In multi-word names, the first letter of each of the second and following words should be capitalized.
Here are some examples:

run
runFast
getBackground
getFinalData
compareTo
setX
isEmpty

A method might have the same name as other methods due to method overloading.

Overloading Methods
Methods within a class can have the same name if they have different parameter lists

(there are some qualifications to this that will be discussed in the lesson titled "Interfaces and Inheritance").
Suppose that you have a class that can use calligraphy to draw various types of data (strings, integers, and so on) and that contains a method for drawing each data type.

```java
public class DataArtist {
    ...
    public void draw(String s) {
        ...
    }
    public void draw(int i) {
        ...
    }
    public void draw(double f) {
        ...
    }
    public void draw(int i, double f) {
        ...
    }
}
```

You cannot declare more than one method with the same name and the same number and type of arguments, because the compiler cannot tell them apart.

The compiler does not consider return type when differentiating methods, so you cannot declare two methods with the same signature even if they have a different return type.
Note: Overloaded methods should be used sparingly, as they can make code much less readable.

Providing Constructors for Your Classes

A class contains constructors that are invoked to create objects from the class blueprint.

Constructor declarations look like method declarations—except that they use the name of the class and have no return type.

For example, Bicycle has one constructor:

```java
public Bicycle(int startCadence, int startSpeed, int startGear) {
    gear = startGear;
    cadence = startCadence;
    speed = startSpeed;
}
```
To create a new `Bicycle` object called `myBike`, a constructor is called by the `new` operator:

```java
Bicycle myBike = new Bicycle(30, 0, 8);
```

`new Bicycle(30, 0, 8)` creates space in memory for the object and initializes its fields.

Although `Bicycle` only has one constructor, it could have others, including a no-argument constructor:

```java
public Bicycle() {
    gear = 1;
    cadence = 10;
    speed = 0;
}
```

```java
Bicycle yourBike = new Bicycle();
```

invokes the no-argument constructor to create a new `Bicycle` object called `yourBike`.

Both constructors could have been declared in `Bicycle` because they have different argument lists.
You don't have to provide any constructors for your class, but you must be careful when doing this. The compiler automatically provides a no-argument, default constructor for any class without constructors. This default constructor will call the no-argument constructor of the superclass. In this situation, the compiler will complain if the superclass doesn't have a no-argument constructor so you must verify that it does. If your class has no explicit superclass, then it has an implicit superclass of Object, which does have a no-argument constructor.

You can use a superclass constructor yourself. The MountainBike class at the beginning of this lesson did just that. This will be discussed later, in the lesson on interfaces and inheritance.

You can use access modifiers in a constructor's declaration to control which other classes can call the constructor.

Note: If another class cannot call a MyClass constructor, it cannot directly create MyClass objects.
Passing Information to a Method or a Constructor

The declaration for a method or a constructor declares the number and the type of the arguments for that method or constructor.

```java
public double computePayment(
    double loanAmt,
    double rate,
    double futureValue,
    int numPeriods) {
    double interest = rate / 100.0;
    double partial1 = Math.pow((1 + interest),
                               - numPeriods);
    double denominator = (1 - partial1) / interest;
    double answer = (-loanAmt / denominator)
                    - ((futureValue * partial1) / denominator);
    return answer;
}
```

**Note:** *Parameters* refers to the list of variables in a method declaration. *Arguments* are the actual values that are passed in when the method is invoked. When you invoke a method, the arguments used must match the declaration's parameters in type and order.
Parameter Types

You can use any data type for a parameter of a method or a constructor.

```java
public Polygon polygonFrom(Point[] corners) {
    // method body goes here
}
```

Arbitrary Number of Arguments

You can use a construct called `varargs` to pass an arbitrary number of values to a method.

You use varargs when you don't know how many of a particular type of argument will be passed to the method.

It's a shortcut to creating an array manually (the previous method could have used varargs rather than an array).

To use varargs, you follow the type of the last parameter by an ellipsis (three dots, ...), then a space, and the parameter name. The method can then be called with any number of that parameter, including none.
public Polygon polygonFrom(Point... corners) {
    int numberOfSides = corners.length;
    double squareOfSide1, lengthOfSide1;
    squareOfSide1 = (corners[1].x - corners[0].x)
        * (corners[1].x - corners[0].x)
        + (corners[1].y - corners[0].y)
        * (corners[1].y - corners[0].y);
    lengthOfSide1 = Math.sqrt(squareOfSide1);

    // more method body code follows that creates and
    // returns a
    // polygon connecting the Points
}

You can see that, inside the method, corners is
treated like an array.

public PrintStream printf(String format,
Object... args)

System.out.printf("%s: %d, %s\n", name, idnum, address);

System.out.printf("%s: %d, %s, %s, %s\n", name, idnum, address, phone, email);
Parameter Names

When you declare a parameter to a method or a constructor, you provide a name for that parameter. This name is used within the method body to refer to the passed-in argument.

The name of a parameter must be unique in its scope.

A parameter can have the same name as one of the class's fields. If this is the case, the parameter is said to shadow the field.

```java
class Circle {
    private int x, y, radius;
    public void setOrigin(int x, int y) {
        ...  
    }
}
```

Remember the `this` Keyword? (will be discussed later)
Passing Primitive Data Type Arguments

Primitive arguments, such as an int or a double, are passed into methods by value. This means that any changes to the values of the parameters exist only within the scope of the method. When the method returns, the parameters are gone and any changes to them are lost.

```java
public class PassPrimitiveByValue {
    public static void main(String[] args) {
        int x = 3;

        // invoke passMethod() with x as argument
        passMethod(x);

        // print x to see if its value has changed
        System.out.println("After invoking passMethod, x = " + x);
    }

    // change parameter in passMethod()
    public static void passMethod(int p) {
        p = 10;
    }
}
```

Output:

After invoking passMethod, x = 3
Passing Reference Data Type Arguments

Reference data type parameters, such as objects, are also passed into methods by value. This means that when the method returns, the passed-in reference still references the same object as before. However, the values of the object's fields can be changed in the method, if they have the proper access level.

```java
public void moveCircle(Circle circle, int deltaX, int deltaY) {
    // code to move origin of circle to x+deltaX, y+deltaY
    circle.setX(circle.getX() + deltaX);
    circle.setY(circle.getY() + deltaY);

    // code to assign a new reference to circle
    circle = new Circle(0, 0);
}
```

Let the method be invoked with these arguments:

```java
moveCircle(myCircle, 23, 56)
```
Inside the method, circle initially refers to myCircle. The method changes the $x$ and $y$ coordinates of the object that circle references (i.e., myCircle) by 23 and 56, respectively. These changes will persist when the method returns.

Then circle is assigned a reference to a new Circle object with $x = y = 0$. This reassignment has no permanence, however, because the reference was passed in by value and cannot change. Within the method, the object pointed to by circle has changed, but, when the method returns, myCircle still references the same Circle object as before the method was called.