1 What is a Class, Again?

After the introduction, we have delayed the discussion of classes up to this point. This is because it is rather difficult to go into the details of defining classes without knowledge of the basic programming structures in Java. You have also seen a solid example of a class now, which is the `String` class. So, now you are ready to understand the notion of a class.

2 A `Circle` Class

We will try to define a `Circle` class. It will be a class of circles in a plane. First, it must have properties, which will be stored in what is called "data members"[1] Now, what properties would a circle have? In other words, what defines a circle? A little thought reveals that the coordinates of the center, and the radius are all that are required to define a circle. Thus, we begin the definition of the `Circle` class as follows:

```
public class Circle {
    private double radius;
    private double centerX;
    private double centerY;
}
```

You do understand what `double` stands for, but `private` should be a mystery. It is not a huge mystery, though. All it means is that only methods of the `Circle` class are allowed to access those variables. We will see soon how that works, so don't worry about it too much right now.

The next thing we can do is define methods for our new `Circle` class. For example, we may define a method for finding the are of our circle. Here is how that would look:

```
public class Circle {
    private double radius;
    private double centerX;
    private double centerY;

    public double getArea() {
        return Math.PI * radius * radius;
    }
}
```

The `public` specifies that the `getArea()` method can be called externally, not just from the `Circle` class. The `double` indicates that this method will return a value of type `double`. This method does not take any arguments, and therefore the pair of parentheses following the method name are empty.

We could add another method that moves our circle. That would make the class look as follows:

```
public class Circle {
    private double radius;
```

[1] "Data members" are also called "fields", or sometimes even "properties".

```
    private double centerX;
    private double centerY;

    public double getArea() {
        return Math.PI * radius * radius;
    }

    public void move(double moveX, double moveY) {
        centerX += moveX;
        centerY += moveY;
    }

}
```

This time our method takes two arguments: `moveX`, which is of type `double`, and `moveY`, which is of type `double` as well. The keyword `void` expresses the fact that our method returns *no value at all*. What this method does is, just move the center of the circle by the offsets given as parameters.

3 Constructors

Assuming we have a class definition, how do we create objects of this class? Somewhere else in the program (usually some other class), there will be a statement such as:

```
Circle c;
c = new Circle();
```

Note that, just as with the `String` class, the declaration does *not* create the actual object. The object itself must be created using the `new` operator. Note the pair of parenthesis after the class name, as if we are calling a method. As a matter of fact, we *are* calling a method. It is called the *constructor*. At this point, you should be saying "But we never defined that method!", and you are right. We did not. But, if you don't have a constructor, Java will create a *default constructor* for you, implicitly. What the default constructor does is initialize all data members to zero. Let us write an explicit constructor for this class:

```
public class Circle {
    private double radius;
    private double centerX;
    private double centerY;

    public Circle() {
        radius = 1.0;
        centerX = 0.0;
        centerY = 0.0;
    }

    public double getArea() {
        return Math.PI * radius * radius;
```

```
    }

    public void move(double moveX, double moveY) {
        centerX += moveX;
        centerY += moveY;
    }

}
```

Note that the constructor does not have a return type, and *always* carries the same name as the class. Our constructor, when called as `new Circle()` will create a unit circle centered at the origin.

We can also have constructors that take arguments. It is also possible to have multiple constructors, as long as they have different argument lists. Here is an example that lets us create a circle of any size centered anywhere:

```
public class Circle {
    private double radius;
    private double centerX;
    private double centerY;

    public Circle() {
        radius = 1.0;
        centerX = 0.0;
        centerY = 0.0;
    }

    public Circle(double cX, double cY, double r) {
        centerX = cX;
        centerY = cY;
        radius = r;
    }

    public double getArea() {
        return Math.PI * radius * radius;
    }

    public void move(double moveX, double moveY) {
        centerX += moveX;
        centerY += moveY;
    }

}
```

The call to this new constructor would look like:

```
Circle c = new Circle(1.0, 2.0, 5.0);
```

4 The `this` Variable

In every method of an object, there is a variable that is defined by default: `this`. This refers to the object whose method is being called. Whenever you refer to a data member of a class, use of `this` is implied. For instance we could write the `move()` method as follows:

```
public void move(double moveX, double moveY) {
    this.centerX += moveX;
    this.centerY += moveY;
}
```

5 The Finer Points of Parameters

Parameters to methods can be both basic types (`int`, `char`, `double` ...) or objects (`String`, `Circle` ...). However, the conventions used when passing them as parameters are different.

When you pass a basic type as a parameter, a *copy* of it is passed to the method. Any changes made within the method to the parameter will *not* affect the original variable. This is known as *passing by value*.

In contrast, when you pass an object as a parameter, no copy is made. (Note that arrays are, in fact, objects.) Instead a new *reference* which points to the same object in memory, and passed to the method. So, any changes made to the *object* using the reference will be made to the same object in memory, and changes will be permanent. However, if you do operations that affect the *reference*, such as making it point to some other object in memory, the original variable will not be affected.

6 `final` Parameters

If you want to make sure that an object passed to a method does not get modified within that method (i.e. is used "read-only"), you prepend the word `final` to the beginning of the parameter declaration. If you attempt to modify a parameter which is declared as `final` within the method body, the compiler will complain.

7 `static` Variables

You can prepend the qualifier `static` to a variable declaration within a class. In that case, that becomes a *static* variable, which means there is only one instance of that variable for the class. In other words, a copy is *not* created per object.

8 `static` Methods

Prepending the qualifier `static` to a method will cause that to become a *static* method. This means that the method can be called even when there are no objects of that class present. A static method can be called by using the class name instead of an object reference. We have already done this in the case of the `Math` class. The `main()` method is also static, since it is called by the Java Virtual Machine without ever creating an instance of the class in question.

9 Packages in Java

Packages are very important and central to the way Java operates. They provide a uniform way of naming and locating classes. Packages are also very closely interconnected with the directory structure of classes on disk. We will see how all this works.

Up to now, all the classes we wrote were in the *default* package, which has no name. Now, let us see how we can put, say, the geometry classes in a package. We will call the package simply `geometry`.[2] But, up to now everything we did was in two dimensions. So it would be a good idea to to create a *subpackage* called `twodim` and place our `Circle`, `Point`, and `Line` classes in it.

Let us start with the `Point` class. First, we need to specify within `Point.java` the package name. This is done with the `package` statement, which must come before anything else in a file. The file will look like the following:

```
package geometry.twodim;

public class Point {
// Class definition... Yada yada.
}
```

Note that package and subpackage names are separated by dots. (There can be as many levels of packages as you like.) Now, we must also put this file in its right place. To do that, we must create a directory called `geometry`. And then, inside it, create a directory called `twodim`. And then, we must put `Point.java` in that directory.

This begs the question: Where do we create these directories? The answer is, in a directory contained in your *classpath*. The classpath is read from the environment variable `CLASSPATH`, or it can be specified on the command line when running Java. with the `-classpath <classpath>` option. This is in fact, how Java locates your classes.

10 The Java Class Library Packages

Java provides a vast library of classes for different purposes. Below is a table of some packages and their contents:

Package Name	Content
`java.lang`	Contains classes used in most programs.
`java.io`	Classes used for input/output.
`java.awt`	Contains GUI classes.
`java.applet`	Classes required for writing applets.
`java.util`	Contains “utility” classes.
`java.sql`	Contains classes to handle database access.

In order to use classes that are in `java.lang`, you need not do anything; those classes are available in your program automatically. However, in order to use classes in other packages, you need to do one of two things:

[2]Note that this is not a very good package naming. Someone else might very well come up with the same package name. Sun suggests that everyone name their packages by using their internet domain names in reverse order. As such, we could name the geometry package `tr.edu.boun.phys495.geometry`. But for now, let us stick with simpler names.

Use the full class name: The "full class name" is the name of a class containing the full package name. For example, the full class name of the `String` class we have been using is not `String`, but rather `java.lang.String`. So, if you need to use the `Applet` class which is in `java.applet`, you need to specify `java.applet.Applet`.

Use the `import` statement: If you want to refer to classes with their short names, you can use the `import` statement to tell the compiler to "import" classes into your program. Note that `import` statements come *after* the `package` statement, but before anything else in a file. There are two ways of doing this:

1. `import` a single class:

```
import java.applet.Applet;
// This imports only the applet class.
// Now you can use it only by typing Applet.
```

2. `import` all classes in a package:

```
import java.applet.*;
// This imports all classes in java.applet.
// You can refer to any class in java.applet
// by its short name now.
```