Defining Classes

In our example programs, we have worked extensively with many classes and methods in the Java Application Programming Interface (API). In this section, we will learn how to define our own classes. We define a class usually because we want to instantiate and operate on objects of that class. So, part and parcel of defining classes is devising a useful set of methods to perform operations on objects of that class.

The ability to extend a class is a fundamental concept in object-oriented programming. A class that extends another "inherits" properties from that class. Inheritance yields properties such as encapsulation, polymorphism, method overloading, etc., and each has a special role in shaping the Java language. We will visit each of these topics shortly.

Our First Class

Let's begin by defining a simple class that represents a city.[1] A `City` object will have two variables: a string representing the name of the city and an integer representing the population of the city. The following is a bare-bones definition of the `City` class:

```
//definition of the 'City' class (bare bones!)
class City
{
   String name;  // name of the city
   int pop;      // population of the city
}
```

To test our `City` class, we could embed it in a test program:

```
public class CityTest
{
   public static void main(String[] args)
   {
      City c = new City();
      System.out.println(c);
   }
}

class City
{
   String name;
   int pop:
}
```

If the code above is placed in a file called `CityTest.java`, it will compile and execute without errors. The output will look something like this:

[1] You may find the title of this section unusual. To be sure, this is not our "first class", because every Java program presented in this book is itself a class that we defined. Hopefully, the flow of the discussion is appropriate, given our experience with the Java language. A later section will explore the question “when is a class a program?”..

```
City@e9d8d771
```

Now, this is pretty useless, but don't despair. The very fact that this code compiles, executes, and generates output means something is taking place that we should understand. Indeed, there are a few features in the program worth thinking about. First, we successfully instantiated a `City` object named `c` using a constructor method named `City()`. Second, we used the object variable `c` as an argument in the `println()` method — and something printed! Note that although we defined a class named `City`, we did not define a constructor method named `City()`. This is legal, but not recommended. The effect is to instantiate a `City` object with its variables initialized to default values appropriate for the type of each variable.

We managed to print the `City` object `c`, and this is also unusual, given our previous experience. Usually we print string messages or values held in variables, but not objects (except `String` objects). As it turns out, it is perfectly legal to pass an object variable — in fact, *any* object variable — to the `println()` method. What gets printed is a string representation of the object. So, we might ask, how is this string representation generated? It is generated automatically from a method called `toString()` in the `Object` class. As we noted in discussing Java's class hierarchy in Chapter 4, all classes are subclasses of the `Object` class, and all classes inherit properties, such as methods, from the `Object` class. In terms of our `City` class, this simple relationship is shown in Figure 1.

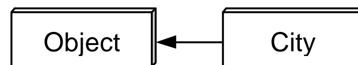

Figure 1. The `City` class is a subclass of the `Object` class

So, the `toString()` method of the `Object` class generated the string that was printed. If we include a `toString()` method in the definition of the `City` class, it is used instead of the `toString()` method in the `Object` class. In effect, one method "overrides" the other. In the absence of this, however, the basic `toString()` method in the `Object` class is used, and you can see the crude effect in the output from our hypothetical program. The string representation of the `City` object `c` shows the name of the class followed by an obscure notation that is of no practical use to us.[2]

Let's develop the `City` class further, into a useful and general-purpose class. For reasons just noted, it a good idea to define a constructor method and a `toString()` method. But, that's not all. The variables `name` and `pop`, as defined above, are public variables. In the vast majority of cases, this is a bad approach to designing a class. We want objects of the `City` class (or any other class) to be well protected. Since it is *our* class, it is OK for us to have control over, and modify, its behaviour, but we do not want to grant similar control to programmers using the class. This is true of all classes in the Java API. The Java designers at Sun Microsystems can modify the classes in the Java API, but we cannot. Our only access is through the public methods and data fields provided by the designers. These are our "interface" to the classes.

In our re-vamped definition of the `City` class, we'll use the `private` modifier for the variables `name` and `pop`. As private variables, they are accessible by methods within the class, but they not accessible by programs that use the class. In fact, this access is so restrictive, we no longer call them variables; they become *fields* — private data fields. Access to the fields is provided by

[2] The characters following the @ symbol are the hexadecimal hash code used to retrieve the object.

"public" methods specifically designed to allow programmers to retrieve and/or alter these fields. The access is highly controlled: If no public method exists to alter a field, then it cannot be altered. Plain and simple.

There is a structural change required as well. If we intend to use the `City` class in other programs, it must reside in its own file and it must be "public". So, we'll put the definition of the `City` class in a file called `City.java` and the code to test the class in a separate file. Figure 2 shows a much-improved definition of the `City` class.

```
 1  public class City
 2  {
 3     // define data fields
 4     private String name;     // name of city
 5     private int pop;         // population of city
 6
 7     // constructor method
 8     public City(String name, int pop)
 9     {
10        this.name = name;
11        this.pop = pop;
12     }
13
14     // other methods
15     public String getName()         { return name;  }
16     public int getPop()             { return pop;   }
17     public void setPop(int newPop)  { pop = newPop; }
18
19     public String toString()
20     {
21        return "City [name: " + name + " pop: " + pop + "]";
22     }
23  }
```

Figure 2. Definition of the `City` class in the file `City.java`

The 23 lines of `City.java` contain all the basic ingredients of a Java class definition. It contains two data fields, `name` and `pop`, and five methods, `City()`, `getName()`, `getPop()`, `setPop()`, and `toString()`. Collectively, the data fields and methods are referred to as *members* of the class.

The definition of the `City` class begins with the class signature:

```
public class City
```

Nothing fancy here, except to note that the visibility modifier "public" is essential for the class to be useable by other classes. Within the definition of `City`, the two data fields `name` and `pop` are defined in lines 4-5. The visibility modifier "private" means the fields are accessible only within the class.

The constructor method is defined in lines 8-12. To construct a `City` object, two arguments are passed: the name of the city and its population. A typical use of the constructor might be

```
City smallTown = new City("Pleasantville", 150);
```

The arguments are assigned to `name` and `pop` in lines 10 and 11, respectively.

Four other methods are defined in lines 15-22. All are "instance" methods (as opposed to "class" methods), and they provide the sorts of services deemed reasonable for objects of this class. The

getName() and getPop() methods simply return the value of the name and pop fields. These are called *accessor* methods because they provide access to data fields in the object.

The setPop() method sets a new population for a City object. Since the effect is to change a data field within the object, it is called a *mutator* method.

The toString() method is a useful override method to generate a string representation of the object. Since these are instance methods, they are called through instances of the City class — namely, by appending the method to a City object using dot notation:

```
String s = smallTown.getName();
smallTown.setPop(175);
int newPop = smallTown.getPop();
```

When the City.java file is compiled, the file City.class is created. However, the City.class file cannot be executed because it does not have a main() method.[3] So, to test the City class, we use a separate file called CityTest.java (see Figure 3).

```
 1  public class CityTest
 2  {
 3     public static void main(String[] args)
 4     {
 5        // instantiate a City object named 'homeTown'
 6        City homeTown = new City("Toronto", 3500000);
 7
 8        // retrieve and print the 'name' field
 9        String s = homeTown.getName();
10        System.out.println("I live in " + s);
11
12        // retrieve and print the 'pop' field
13        int count = homeTown.getPop() - 1;
14        System.out.println("So do " + count + " others");
15
16        // change the value of the 'pop' field
17        homeTown.setPop(37500000);
18
19        // output updated fields for 'homeTown' object
20        System.out.println("Now there are " + homeTown.getPop() +
21           " people in " + homeTown.getName());
22
23        // print the 'homeTown' object (uses toString() method)
24        System.out.println(homeTown);
25     }
26  }
```

Figure 3. CityTest.java

The CityTest program generates the following output:

[3] Actually, the City class does have a main() method, but this is of no concern to us here. We will say more about this in the section called "When is a class a Program?"

```
I live in Toronto
So do 3499999 others
Now there are 3750000 people in Toronto
City [name: Toronto pop: 3750000]
```

There is nothing in the `CityTest` program to surprise us, given our extensive use of classes in the Java API in previous programs. A `City` object named `homeTown` is declared in line 6 and assigned a reference to a `City` object, as returned by the constructor method. The fields are initialized with "Toronto" for the city's name and "350000" for the population. Over the rest of the program, instance methods of the `City` class are called to retrieve or set values in the fields of `homeTown`.

Note that the string representation of the object printed in line 24 was generated by the `toString()` method in the `City` class, not the `toString()` method in the `Object` class, the former having effectively overridden the latter.

Encapsulation

We noted earlier that the "private" modifier for `name` and `pop` precludes external access to these fields. Thus, the internal implementation of the class is hidden. Access to the data is through methods, rather than through the variables themselves. This process of data hiding is known as *encapsulation*, and it is a defining characteristic of object-oriented programming languages like Java. The idea is illustrated in Figure 4.

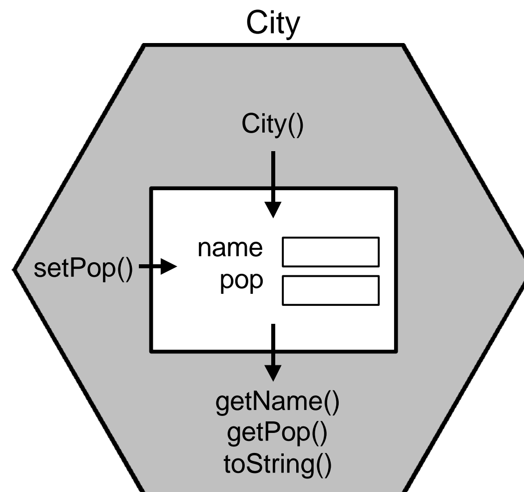

Figure 4. Encapsulation. The data fields in a `City` object are hidden. Access is through methods of the `City` class.

Note the direction of arrows in Figure 4. Only the constructor and `setPop()` have the ability to change fields in a `City` object. The `getName()`, `getPop()`, and `toString()` methods can access the fields, but the operation is strictly "reading" a field's contents. From the programmer's perspective, the methods of a class are the "interface" through which the data are accessed. As noted earlier, methods that read field contents are *accessor* methods; methods that change field contents are called *mutator* methods.

Method Overloading

A Java method is identified by a *signature*, giving the name of the method, the type of the value returned, and the name and type of arguments passed to the method. There can be more than one method with the same name, provided the type or number of arguments varies. This feature is known as *method overloading*. Resolving which to use is the job of the compiler, and, as long as the choice is unambiguous, it's pretty straightforward. This is similar to operator overloading, which we met earlier. (The "+" operator means "addition" if both operands are numbers, but it means "concatenation", if at least one operand is a string.)

Constructors are often overloaded, and numerous examples are found in the Java API documentation. Let's illustrate method overloading with the `City` class. It is possible we may wish to instantiate `City` objects without specifying populations. Since we have a `setPop()` method, the population could be set after the objects are instantiated. For our example, the city names could be read from one file, while the populations are read from another.

To add a "name-only" constructor for the `City` class, we simply add the following method in the class definition:

```
// constructor for City objects specifying a name only
public City(String name)
{
   this.name = name;
   this.pop = 0;
}
```

With this as part of the class definition, the following are both legal instantiations of `City` objects:

```
City windy = new City("Chicago");
City rainy = new City("Vancouver", 250000);
```

In this case a `City` object named `windy` is instantiated and its name field is initialized with "Chicago". The `pop` field was not provided so the name-only constructor is used, and pop is initialized with zero.

In some cases, it may be useful to hard-code a special initialization value in the constructor, for example

```
public City(String name)
{
   this.name = name;
   this.pop = -1;
}
```

Having `pop` initialized to -1 may serve as a useful "flag" within a program to check if the `pop` field has been set using the `setPop()` method.

The notes above are essential in our study of Java, since defining and using our own classes prepares us to embark on more ambitious programming projects. It is strongly urged that you review these notes if you are at all unsure about defining classes.