

User-Defined Classes and Abstract Data Types

(CISS-110 William Jojo)

Overview

Up to this point we have been solving problems using *objects*. Recall that an object is an instance of a *class*. Classes enable us to combine data and methods to operate on that data as a single, manageable type. This combination of data and methods is known as *encapsulation*. For every class, the data and methods of the class are known as *members*.

A class can be defined as simply as:

```
public class MyClass
{
    public int x;
}
```

The modifiers `public` and `private` we have seen before and these modifiers will be used extensively throughout this section. Another modifier, `protected`, will be discussed in a separate document. All three of these are reserved words in the Java language.

`MyClass` has a single variable named `x` which is of type `int` and is a `public` declaration. This means we can access the variable outside of the class. Being that `x` is a *member* of `MyClass`, it is also known as a *field*. Data members of classes are also called *fields*.

Remember that `MyClass` does not actually allocate any memory. This is simply a definition or template for objects that will be created later. Only when we *instantiate* an object is memory actually allocated. Recall that our *reference variables* contain the *addresses* of our instantiated objects.

Classes

Using the class definition from above, we know that anything declared in the class as `public` can be accessed from outside the class. We know this to be true already because we can use `pow()` from the `Math` class and `nextInt()` from the `Scanner` class. If these were not declared as `public`, in other words if they were declared as `private`, we could not use them in their respective classes. Members declared as `private` can only be accessed from within the class itself.

When declaring members in a class those items that are named constants or variables are declared just as you've always done. This is also true when defining methods that are members of the class. In addition, when defining your methods in a class, you have direct access to the fields (data members) of class. It does not matter if the fields are declared as `public` or `private` – the methods of the class have direct access.

To solidify the class concepts described thus far and to introduce new concepts, let us define a class called `Person`. This class will have a given name, surname (last name) and age. Since not everyone has a middle name and the concept of a middle initial is not always simple (a person may have more than one middle name) we will forgo the entire middle name concept for this example. The tradition of some Asian cultures where the family name is first and the given name last is also not addressed in this example.

The beginning of the class looks a bit like the following:

```
public class Person
{
    private String givenName;
    private String surname;
    private int age;
    ...
}
```

Now that we have the basic data types, let us now consider how we will access and modify the contents of this class. Remember that eventually we will be instantiating objects of this type and since the fields are private, we cannot directly access these values outside of the class.

The fields that do *not* have the `static` modifier are known as *instance variables*. We call these fields instance variables because each object that we instantiate, that is, each instance of the class, will have its own set of these variables.

Consider for a moment that we made everything in the class `public`. Why then create a separate user-defined class if everything is `public`? We could just create variables in some `main()` method and arrange to have them passed as parameters to a group of methods that we later define. Well, that is actually part of the point. Beside the use of encapsulation as a means of tying the data together with the methods that maintain the data, is the detail of not having to pass these data items to the methods. By the use of encapsulation, we have tied the data to the operations *and* simplified the mechanism by which we make the data available to the methods. Recall that with user-defined methods we were concerned about the number of parameters and their respective types. Since the data and the methods operating on the data are in the class we eliminate the need to pass the data as parameters. Remember that the methods of a class already have direct access to the fields.

As a final note on the use of private fields, we force the use of this class to follow strict guidelines. No program may directly manipulate the fields – the program must use the operations provided to gain access to this data.

Let us now define some of the operations we need to perform on the private fields of the `Person` class. These operations are itemized below:

1. Set the given name.
2. Return the given name.
3. Set the surname.
4. Return the surname.
5. Set the age.
6. Return the age.
7. Increment the age.
8. Compare two persons for equality.
9. Copy a person.
10. Return a copy of a person.
11. Print a person.

For each of these 11 operations, we will construct a *method*. It is often difficult to determine if the methods of a class should be `public` or `private`. The methods for the 11 operations will be `public` since they all need to be accessed from outside the class. We will discover later that operations 1, 3, 5, 7 and 9 will be mutator methods and 2, 4 and 6 are accessor methods.

The methods for these operations are shown in *Table 1*.

Methods for the Person class
public void setGivenName(String s); Set the given name.
public String getGivenName(); Return the given name.
public void setSurname(String s); Set the surname.
public String getSurname(); Return the surname.
public void setAge(int y); Set the age in years.
public int getAge(); Return the age in years.
public void incAge(); Increment the age by one year.
public boolean comparePerson(Person otherPerson); Compare this person to another person. Return true if all fields are identical.
public void makeCopy(Person otherPerson); Copy the contents of otherPerson into this object.
public Person getCopy(); Returns a new Person object with a copy of the current object contents.
public String toString(); Print a person.

Table 1: Methods that represent the operations allowed in the Person class.

Constructors

Before we delve into the details of the methods in *Table 1*, let us first consider *instantiation*. Recall that we create or instantiate objects with the `new` reserved word and a *constructor* is called when the object is created.

Constructors are special methods of classes. These constructors have the same name as the class and are intended to help initialize the instance variables of an object. Every class has at least one constructor, even if you do not define one.

Like methods, constructors may take a list of parameters. The constructor with no parameters is known as the *default constructor*. If you do not define any constructor at all, the default constructor will be created for you by Java. In addition, if you define a constructor and *do not* provide a default constructor, Java *will not create the default constructor*.

Since constructors are similar to methods, they may be overloaded. An overloaded constructor must differ in some way from the other constructors. This is done by either having a different number of parameters or by differing types for at least one position. Their *signatures* must be different.

If you do not elect to set values for the instance variables via the class constructor(s), then each is initialized with a *default value*. The Java documentation defines default values as a type appropriate zero and `null` for reference variables.

We will define two constructors for the `Person` class. One will be the default constructor and the other will have three parameters representing the instance variables. These constructors are shown in *Table 2*.

Constructors for the Person class
<pre>public Person();</pre> Default constructor initializing <code>givenName</code> and <code>surname</code> to empty strings and <code>age</code> to zero.
<pre>public Person(String g, String s, int a);</pre> Constructor to initialize <code>givenName</code> to <code>g</code> , <code>surname</code> to <code>s</code> and <code>age</code> to <code>a</code> .

Table 2: The list of Constructors for the Person class.

Here is a list of properties with respect to constructors:

- The name of the constructor and the name of the class are the same.
- A class can have many constructors. All constructors have the same name and therefore can be overloaded.
- A constructor is neither a `void` method nor a value-returning method.
- Overloaded constructors must differ in either the number of parameters or at least one parameter must be a different type when the number of parameters is the same.

Variables and Instantiation

With the framework of the class in place, we will now work with a couple of reference variables. We will start with a refresher. Recall that:

```
Person firstPerson;  
Person secondPerson;  
  
firstPerson = new Person();  
secondPerson = new Person("Bob", "Smith", 36);
```

Can also be combined into:

```
Person firstPerson = new Person();  
Person secondPerson = new Person("Bob", "Smith", 36);
```

Figure 1 shows the reference variables and their respective objects. As you can see, each was initialized based on the rules defined above. That is, the default constructor would initialize the strings to the empty string and the age to zero. The alternate constructor would use the three parameters provided to initialize the object.

Recall that the dot (.) is the *member access operator*. This operator is used to access named constants, methods and instance variables within an object. However, the instance variables listed for this class are `private`. Therefore, the following statements are *illegal* with respect to the `Person` class:

```
firstPerson.givenName = "Steve"; // illegal
firstPerson.surname = "Jones";   // illegal
firstPerson.age = 35;            // illegal
```

We are only allowed to access `public` and `public static` members of the class from outside the class. Recall that `public` members are accessed on an instance of the class (object) while `public static` members can be access on the class itself.

Since the previous set of statements have been proven invalid, we are forced to use the methods provided in the first place. Remember that this is by design. The following statements are completely legal:

```
firstPerson.setGivenName("Steve");
firstPerson.setSurname("Jones");
firstPerson.setAge(35);
```

We now have the layout depicted in Figure 2.

Be careful with the use of the assignment operator with reference variables. Consider the following statement:

```
firstPerson = secondPerson;
```

On the surface you might expect to have the contents of the `secondPerson` object copied in the `firstPerson` object. Remember that with reference variables, the use of the assignment operator assigns *addresses*, not content. The result of the above statement is shown in Figure 3. This particular form of copying is know as *shallow copying*. Shallow copying is used when you intend to have more that one reference variable point to the same object.

If your intentions are to copy the contents, you need to perform a *deep copy*. A deep copy results

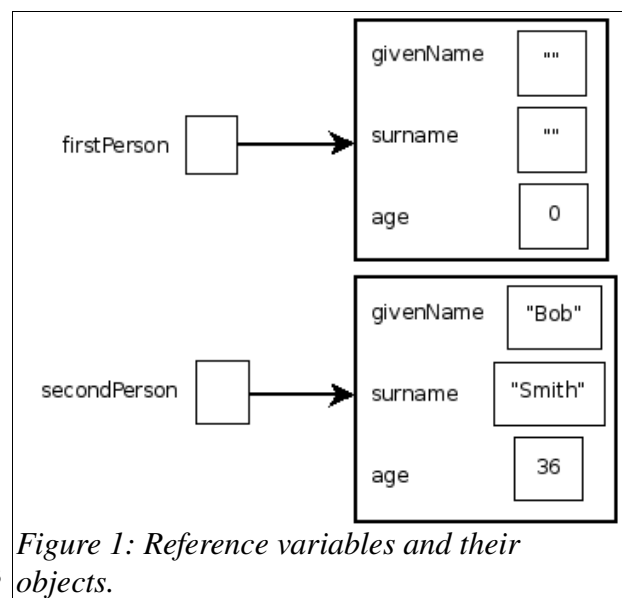


Figure 1: Reference variables and their objects.

in each reference variable having the same data, but pointing to *different* objects.

Figure 4 shows the result of a deep copy.

We can achieve this deep copy by one of two methods:

- We can either copy the contents of an object into an existing object by using the `makeCopy ()`.
- Create a new object from an existing object copying the contents of the original object in the new object.

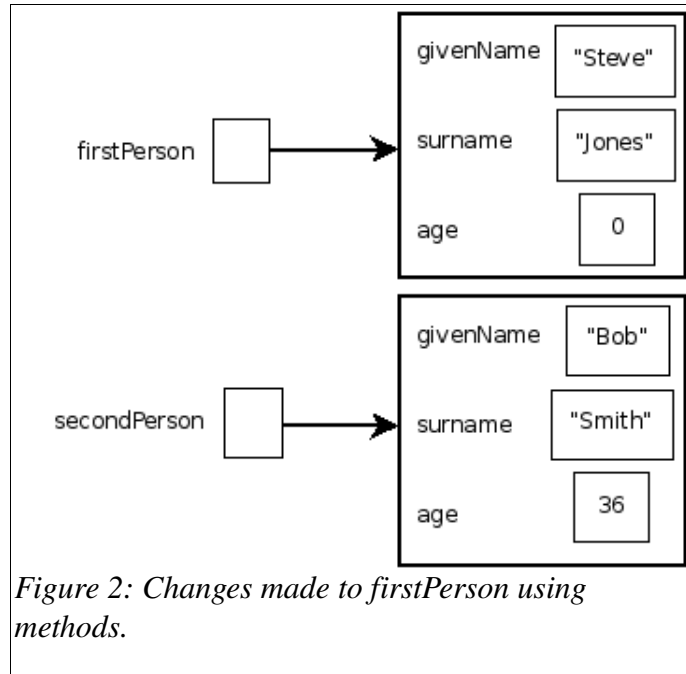


Figure 2: Changes made to firstPerson using methods.

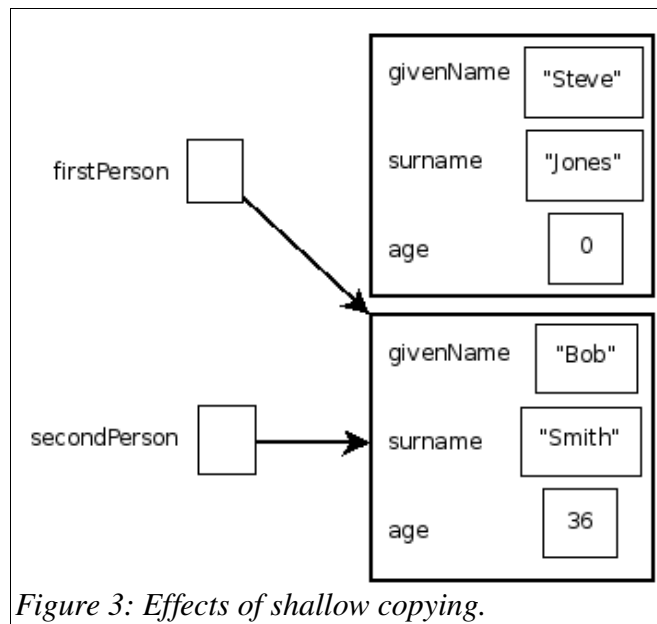
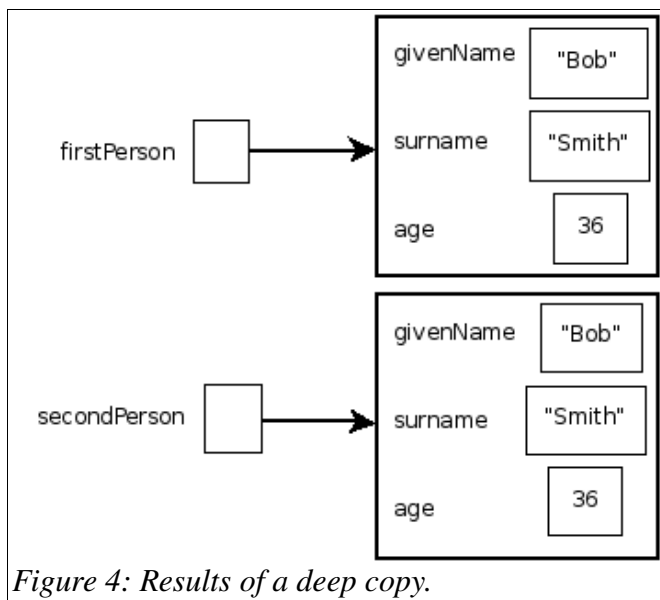


Figure 3: Effects of shallow copying.



To copy the contents of an object into an existing object, we can use the following statement:

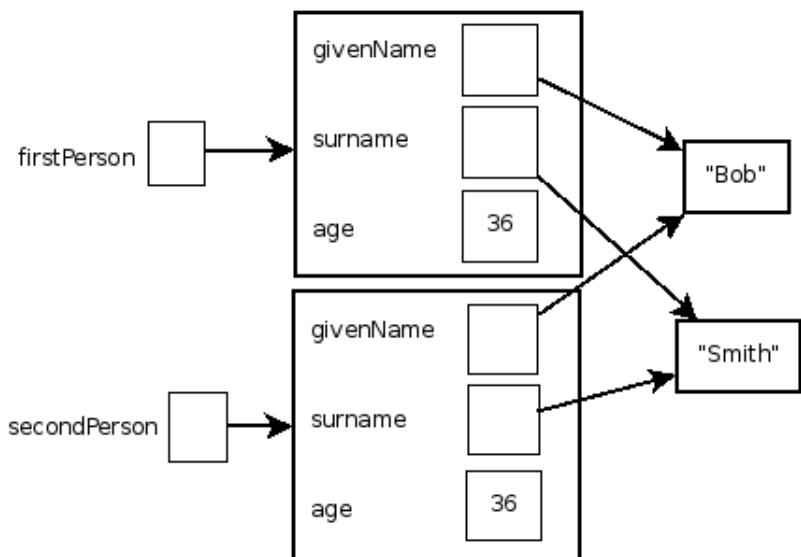
```
firstPerson.makeCopy(secondPerson);
```

The other method of creating a new object from an existing object is done with the following statement:

```
firstPerson = secondPerson.getCopy();
```

Let us discuss one final note about the depiction of `firstPerson` and `secondPerson` and their respective instance variables. These instance variables `givenName` and `surname` are also reference variables and point to objects. Technically, *Figure 4* should look more like *Figure 5*.

In *Figure 5* we see that although we've duplicated or copied the objects, ultimately the String objects are shared between the objects. Recall that this is due to the fact that Java only uses one copy of the string among many references since they are



immutable objects.

```
public Person()
{
    givenName = "";
    surname = "";
    age = 0;
}

public Person(String g, String s, int a)
{
    givenName = g;
    surname = s;
    age = a;
}
```

Example 1: Constructors for the Person class.

Constructors and Methods of the Person Class

Now that we have spent some time describing the constructors and methods of the `Person` class and how they can be used, let us now define all of the constructors and methods in Java.

Let us begin with the constructors. *Example 1* shows the current set of constructors. As discussed previously, the default constructor will initialize `givenName` and `surname` to the empty string and `age` to zero.

Now is a good time to discuss the differences between accessor and mutator methods of a class. An *accessor method* retrieves or accesses the values of instance variables whereas a *mutator method* modifies them. The methods `getGivenName()`, `getSurname()` and `getAge()` are accessor methods while the methods `setGivenName()`, `setSurname()`, `setAge()` and `incAge()` are mutator methods.

Remember that the point of this level of encapsulation is to manage this object and other object individually using the same set of methods and constructors. The accessor methods are shown in *Example 2*, while the mutator methods are shown in *Example 3*.

The remainder of the methods are shown in *Example 4*.

```
public String getGivenName()
{
    return givenName;
}

public String getSurname()
{
    return surname;
}

public int getAge()
{
    return age;
}
```

Example 2: Accessor methods of the Person class.

```
public void setGivenName(String s)
{
    givenName = s;
}

public void setSurname(String s)
{
    surname = s;
}

public void setAge(int y)
{
    age = y;
}

public void incAge()
{
    ++age;
}
```

Example 3: Mutator methods of the Person class.

```

public boolean equals(Person otherPerson)
{
    return ( ( age == otherPerson.age) &&
            ( givenName.compareToIgnoreCase(otherPerson.givenName) == 0 ) &&
            ( surname.compareToIgnoreCase(otherPerson.surname) == 0 ) );
}

public void makeCopy(Person otherPerson)
{
    givenName = otherPerson.givenName;
    surname = otherPerson.surname;
    age = otherPerson.age;
}

public Person getCopy()
{
    Person temp = new Person(givenName, surname, age);

    return temp;
}

public String toString()
{
    return givenName + " " + surname + ". Age " + age + ".";
}

```

Example 4: Remaining methods of the Person class.

The `equals()` method is called on an instance of the `Person` class to compare with another instance of `Person` passed as a parameter. This means we would use a statement like:

```

if ( firstPerson.equals(secondPerson) )
{
    ...
}

```

The `equals()` method returns `boolean` so we can use it easily in a conditional test as demonstrated above.

Copy Constructor

Recall the `getCopy()` method of `Person` that created a new object and duplicated the values of the instance variables. Since `getCopy()` creates a new object, we could take the step to provide a *copy constructor* for the class that performs the same work, but at the time of instantiation rather than having a separate method to create and perform the copy.

Consider this example:

```
public Person(Person otherPerson)
{
    givenName = otherPerson.givenName;
    surname = otherPerson.surname;
    age = otherPerson.age;
}
```

Like the other constructors, the copy constructor becomes the initializer of the object and uses the `otherPerson` object to populate the new object. Consider the following:

```
Person firstPerson = new Person("Pat", "Connor", 56);
Person secondPerson = firstPerson.getCopy();
```

With the copy constructor this can be rewritten as:

```
Person firstPerson = new Person("Pat", "Connor", 56);
Person secondPerson = new Person(firstPerson);
```

This helps in the understanding of the code since we cannot immediately perceive the intent of `getCopy()` in the first version. In the second version, the use of the copy constructor and the new operator clearly indicates that an object is being instantiated, that `firstPerson` will be used to initialize the object assigned to `secondPerson` and we eliminate the need for a `getCopy()` method.

The `toString` Method

Java provides a default constructor for a class when no constructors are provided by the programmer. In addition, Java will also provide a method called `toString()` if you do not.

The `toString()` method returns a string value when called. The purpose of this method is to allow *objects* to be printed. While that sounds a bit strange, there are plenty of reasons to display an object. For example, when you run the following:

```
int x = 10;
System.out.println("The value of x is " + x + ".");
```

The variable `x` is boxed to the `Integer` class which has a `toString()` method to return a

String object. Only then would we have three String objects to concatenate together for the println() method. Of course, this level of detail has been hidden behind the autoboxing and auto-unboxing nature of Java.

Consider the following statement:

```
System.out.println(firstPerson);
```

Now if we had not provided the toString() method shown earlier, the default value printed for a class by the Java provided toString() method would be:

```
Person@16a8b92
```

This is the name of the class followed by an at sign followed by the hash code for the object. the hash code is used by Java to locate the object in memory and does not represent an actual memory location. Since this particular form of output is rather useless to most, the toString() method provided earlier allows us to produce output similar to:

```
Pat Connor. Age 56.
```

Static Members of a Class

Recall that static methods of a class can be called on the class as there is no need to create an object of the class. For example:

```
double f;  
...  
f = Math.pow(5,3);
```

No Math object was created. We simply called pow() on the class and this method works on any version of Java's JDK. As of JDK 5.0, you may also import statically using an altered import statement and simplifying the code like:

```
import static java.lang.Math.*;  
...  
double f;  
...  
f = pow(5,3);
```

Of course, you will need static import statements for the classes with which you wish to use this simplified style.

You may also create static instances of variables within a class. These too can be accessed on the class without needing to create an object providing they are also declared as

public. Therefore, any public static members may be accessed on the class without needing to create an object.

With that said, variables of a class that are static exist without any instances of class. That is, you can access their values *prior to* instantiating any objects. This is due to the fact that when you instantiate an object, only the non-static variables are created in the object. The static variables are *shared* between all of the objects. Consider the following:

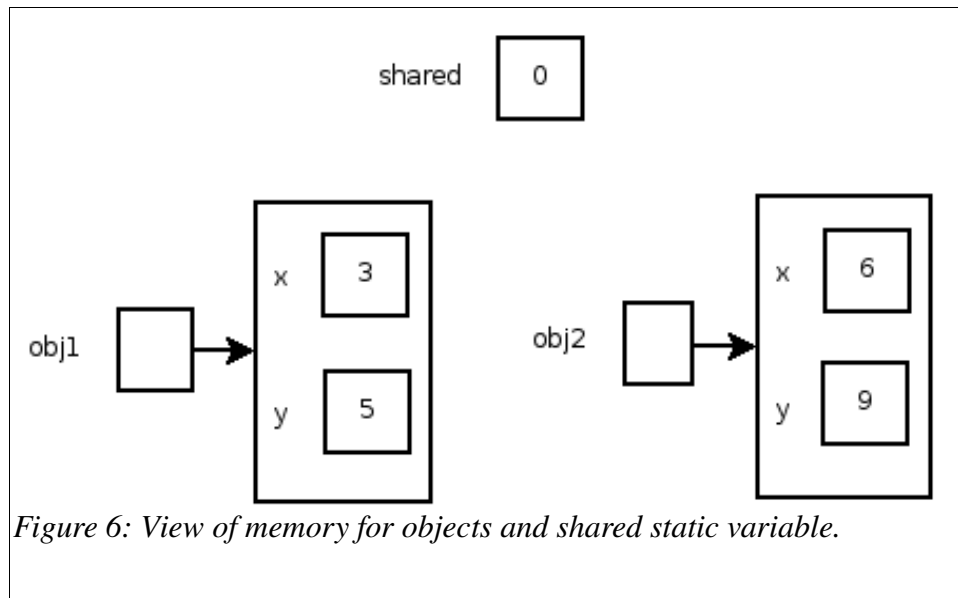
```
public class SharedData
{
    private int x, y;
    private static int shared;

    public SharedData(int xval, int yval)
    ...
}
```

The memory already exists for the static variable `shared` before we instantiate any objects. In addition, `shared` is initialized to their standard defaults, which in this case is zero.

Variables that are static are useful when objects need to share data whereas maintaining this information between objects would be far too involved. *Figure 6* depicts the memory of the following code:

```
SharedData obj1 = new SharedData(3,5);
SharedData obj2 = new SharedData(6,9);
```



The `this` Reference

Whenever Java is executing methods for a class, it needs to know which object it is executing methods on and which instance variables to access. It does this through the implicit use of the reserved word `this`. Every object has access to a reference of itself. Consider the following completed `SharedData` constructor:

```
public SharedData(int xval, int yval)
{
    x = xval;
    y = yval;
}
```

This particular piece of code actually has an implied use of the `this` reference and can be rewritten like the following:

```
public SharedData(int xval, int yval)
{
    this.x = xval;
    this.y = yval;
}
```

We can also use it explicitly to resolve conflicts in naming if the `SharedData` constructor had parameter names that matches the instance variable names as in the following example:

```
public SharedData(int x, int y)
{
    this.x = x;
    this.y = y;
}
```

As you can see, we have removed the ambiguity over the use of `x` and `y`. How else would the compiler know that we are talking about the parameter `x` versus the instance variable `x` simply by using `x`?

Inner Classes and File Scope

The classes we've defined thus far are designed with *file scope*. This means that the class itself is a file. It starts out as a Java source file, then this is compiled into a class file, but cannot be executed on its own since it contains no `main()` method, nor should it.

Once that file is compiled, the class file often lives in the same directory as the programs that reference that class unless the class file is made part of a package which could then be imported by any Java program thereafter.

The class we created during the GUI discussion [*Graphical User Interface (an Introduction)*] for handling the button events was created inside of an existing class. This is called an *inner class* and are typically used for exactly this purpose.