

User-Defined Methods

(CISS-110 William Jojo)

Overview

User-defined methods are methods created by the application writer to support a particular feature or fulfill a specific need that has not already been met through the use of standard objects and their predefined methods. This concept of creating our own methods will be enhanced later as we define methods specifically for manipulating user-defined classes and the objects we will create from those classes.

This discussion predicates the basic concepts of object-oriented design (OOD). Thus far we have used `Math`, `JOptionPane`, `String`, `Scanner` and wrapper classes. We have also used the `System` class with two specific fields, `in` and `out`, which are objects of two other classes, `InputStream` and `PrintStream`.

What is important about this is you were able to use each of these classes and their methods to do work without any additional knowledge of how the work was done or how strings are stored or how the `System.out` standard output object manages to get the characters to the screen.

Each object that we create and store in our reference variables has both relevant data and operations (methods) that can manipulate that data. The `String` objects that we have used contained both the memory to store the characters that make up the string plus the `length` of the string, and methods like `toUpperCase()`, `charAt()`. They are combined together in the same object and this is referred to as *encapsulation*.

Wrapper Classes – Revisited

Recall the `Integer` wrapper class. This and other wrapper classes are used quite frequently to turn primitive types into objects and back again. Consider the following:

```
int x, vol;  
Integer xobj;  
  
x = 15;  
xobj = new Integer(60);
```

Named Constants
public static final int MAX_VALUE = 2147483647;
public static final int MIN_VALUE = -2147483648;
Constructors
public Integer(int num) // Create an Integer object whose initial value is num.
public Integer(String str) // Create an Integer object whose initial value is str converted to an int.
Methods
public int compareTo (Integer otherInteger) // Compares the integer value of the object to the value stored in otherInteger and return 0 if they are equal, less than 0 if this Integer is less than otherInteger and greater than 0 if this Integer is greater than otherInteger.
public int intValue() // Returns the integer value stored in the object.
public double doubleValue() // Returns the integer value stored in the object converted to double.
public boolean equals(Object obj) // Compares the integer value stored in the object to the value stored in obj and returns true if there are the same and false otherwise.
public static int parseInt(String str) // Return the conversion of str to an int.
public String toString() // Return a string representation of the object value.
public static String toString(int num) // Return the value of num as a string.
public static Integer valueOf(String str) // Return an Integer object whose value is derived from str.

Table 1: Recap of the Integer wrapper class.

The variable `x` is simply a primitive type `int`. The reference variable `xobj` contains the address of an object; an instance of `Integer`. The object was *instantiated* by use of `new`. These terms should not be foreign to us. We have seen these words tossed around quite a bit. *Table 1* summarizes some of the details of the Integer wrapper class.

The difference between `x` and `xobj` is that `x` can be used anywhere an integer can be used, such as a simple expression like:

```
vol = x * 40;
```

or

```
if ( x == 45 )  
    vol = 100;
```

The `xobj` variable is an instance of a class and prior to JDK 5.0, would have to be used like so:

```
vol = xobj.intValue() * 40;
```

or

```
if ( xobj.intValue() == 45 )  
    vol = 100;
```

[See the discussion about autoboxing and auto-unboxing for details of changes in JDK 5.0 regarding the use of the wrapper classes.]

An object can almost be used like any primitive type. The requirement to success is that we provide methods to access the data stored in the object. The `Integer` class has *data*, *named constants*, *constructors* and *methods*. The only item we haven't discussed is the *constructor*.

Constructors are methods that are called when we use the `new` operator. Although `new` has been treated as a keyword, it is in actuality a unary operator. This operator has the job of instantiating a class or what we also call creating an object. Constructors are methods that have the same name as the class.

What we are about to embark on is the creation of our own methods that will lead us to building our own classes that include constructors, methods, named constants and of course, our data.

Autoboxing and Auto-Unboxing

The Java language introduced a change to the handling of wrapper classes starting with JDK 5.0. Although the previous examples using the `Integer` wrapper class are quite correct, they were also quite necessary in earlier versions of Java.

Autoboxing is the act of Java automatically calling on the appropriate constructor when assigning a primitive type to a wrapper class reference variable.

```

public class boxing
{
    public static void main(String[] args)
    {
        int x, y;
        Integer xobj, yobj;

        // Simple primitive types
        x = 10;
        y = 15;
        System.out.println("x is " + x);
        System.out.println("y is " + y);
        System.out.println("x * y = " + x * y);
        System.out.println();

        // Wrappers prior to JDK 5.0
        xobj = new Integer(100);
        yobj = new Integer(150);
        System.out.println("xobj is " + xobj);
        System.out.println("yobj is " + yobj);
        System.out.println("xobj * yobj = " +
            xobj.intValue() * yobj.intValue());
        System.out.println();

        // Wrappers with JDK 5.0 and later
        // Autoboxing
        xobj = 1000;
        yobj = 1500;
        // Auto-unboxing.
        System.out.println("xobj is " + xobj);
        System.out.println("yobj is " + yobj);
        System.out.println("xobj * yobj = " + xobj * yobj);
        System.out.println();
    }
}

```

Example 1: JDK 5.0 program demonstrating autoboxing and auto-unboxing.

Auto-unboxing is the extraction of the primitive data from the object so that it can be used in simple expressions. This is demonstrated in Example 1.

You may not appreciate this now, but when you are using Collections or other advanced objects, this simplifies coding considerably. Just be mindful that this makes your code non-portable to previous versions of the Java language.

Note that in *Example 1*, when we change the value of `xobj` from 100 to 1000, we don't actually modify the data contained in the object. A new object is created. This is due to the fact that the wrapper class objects are *immutable*. This means that once the value is set it

cannot be changed. Instead, when the value of the object is changed, the object is actually replaced with a new object with the new value. The `String` class also behaves this way.

Recall that garbage collection happens automatically when we no longer need our objects. Fortunately (or perhaps *unfortunately*) we do not have to clean up the mess that is made by these objects being created and subsequently ignored at a potentially high volume.

User-Defined Methods

Now that we've covered a bit of ground on classes, objects, instantiation, methods, wrappers, autoboxing, constructors and the possible immutability of an object, it is finally time to create our own methods. [Phew!]

Methods in predefined classes allowed us to do many things. We can use the premise of methods in our own code. The premise is this: *anything that we could do one time, we may likely want to do again*. Now, before you go thinking about a loop, consider this: what if you want to do some set of statements now and then 35 lines of code later, do it again?

A loop would not suffice when the desire to do something more than once is not associated with the same point in time. We already know that a method allows us to call upon the code to perform some action whenever we see fit to do so. This flexibility relieves us of the necessity to constantly copy and paste sections of code. Then comes the tedium of subtle modifications to that code if it is not quite suitable for the current need.

Let us use some of the code from *Example 1* to create a new program that will demonstrate the use of methods and why we would desire to create our own.

In *Example 2*, the values of `x` and `y` will change two additional times after the first set of values is assigned. Recall from *Example 1* that the code to display the output was like so:

```
x = 10;
y = 15;
System.out.println("x is " + x);
System.out.println("y is " + y);
System.out.println("x * y = " + x * y);
System.out.println();
```

After the values of `x` and `y` are changed, we would need to copy and paste the four `println()` statements. Instead, we have created an additional static method called `displayxy()`.

This new method will do the work of the four statements without having to repeat them whenever they are required. We also pass the values of `x` and `y` to `displayxy()` each time we use it. These values that are passed are called *parameters*. You've been doing this all along with the predefined methods and as such this mechanism is available to us for our

```

public class uDMETHODS
{
    public static void greeting()
    {
        System.out.println("\nGood day.");
        System.out.println("Let's have fun with integers!\n");
    }

    public static void displayxy(int xval, int yval)
    {
        System.out.println("x is " + xval);
        System.out.println("y is " + yval);
        System.out.println("x * y = " + xval * yval);
        System.out.println();
    }

    public static void main(String[] args)
    {
        int x, y;
        Integer xobj, yobj;

        greeting();

        x = 10;
        y = 15;
        displayxy(x, y);

        x = 100;
        y = 150;
        displayxy(x, y);

        x = 1000;
        y = 1500;
        displayxy(x, y);
    }
}

```

Example 2: Java program demonstrating user defined methods.

user-defined methods.

There is one additional method. The `greeting()` method is used to just display a greeting to the user and has no parameters passed to it.

There are two kinds of user-defined methods:

- Value-returning methods.
- Void methods. (Void methods return no value)

The `main()` method is a void method – we have known this for quite some time due to the `void` keyword to the left of `main(String[] args)`. The methods `greeting()` and `displayxy()` are also void methods.

Here is the output generated by *Example 2*:

```
Good day.  
Let's have some fun with integers!  
  
x is 10  
y is 15  
x * y = 150  
  
x is 100  
y is 150  
x * y = 15000  
  
x is 1000  
y is 1500  
x * y = 1500000
```

Calling Our Methods

The act of using our own methods or predefined methods in a program is known as a *method call*. To call a method is to *invoke* it, or make it perform its work. You will see many pieces of documentation use the terms *call* and *invoke* interchangeably and so will we.

Note that when we defined our `displayxy()` method it is **not called** as a result of its mere presence. This is also true of the `main()` method, that is, the `main()` method was not called until the JRE invoked it as a result of loading our class file which is the bytecode that represents the program we wrote. Remember that the `main()` method is the starting point when we actually run the program.

Therefore, `displayxy()` is not implicitly called. We explicitly call it by using its name in the form of a statement. After each value of `x` and `y` is set, the next statement is:

```
displayxy(x, y);
```

Without this statement in our code, nothing would happen with regard to outputting the values of `x` and `y` that we just set. Just remember this detail when writing your own methods – if you

don't call it, it doesn't happen.

Void Methods

Void methods are the easiest to get a handle on since we've been writing one void method in particular for some time, that is, the `main()` method.

Our `greeting()` and `displayxy()` methods have the *modifier* `static`. This means that the methods can be called on the class and not an instance of the class.

Recall that `pow()` was called on the class like:

```
Math.pow(x, y);
```

And `nextInt()` was called on an instance of the class like:

```
Scanner kb = new Scanner(System.in);  
x = kb.nextInt();
```

The method `pow()` is a static method and therefore can be called *on the class*. Whereas `nextInt()` is not a static method and needs to be invoked on `kb`, or *on an instance* of the `Scanner` class.

Let us look at `displayxy()` in a bit more detail. Each parameter passed must be of type `int`. We know this because of this definition:

```
public static void displayxy(int xval, int yval)
```

This indicates that exactly two parameters (labeled as `xval` and `yval`) will be required. This line also defines our *formal* parameter list. The variables declarations `xval` and `yval` are our formal parameters and are both required to be of type `int`.

Each time we call our method with:

```
displayxy(x, y);
```

We name the *actual* parameters to be used. So the variables `x` and `y` are our actual parameters and `xval` and `yval` are our formal parameters. Another way to think about all of this is we *actually* passed `x` and `y`. We will *formally* address the values passed to `displayxy()` as `xval` and `yval`.

Watch out for making the mistake of treating a method definition as if it were a method call. Although `x` and `y` are integer variables, note that we do not add `int` in the call like:

```
displayxy (int x, int y);    // bad method call!
```

This is not a method call from the compiler's point of view. Rather, it is a poor attempt at a method definition.

Remember that the `displayxy()` method is a void method with a parameter list and the `greeting()` method is a void method with no parameter list.

Method Overloading (a first glance)

The code in *Example 3* takes the majority of code from *Example 1* and *overloads* the `displayxy()` method created in *Example 2*.

Overloading a method allows us to create many methods with the same name, but allow us to define some as void methods while others may be value-returning methods. Overloading also allows us to have varying parameter lists where the number and type of parameters can differ greatly.

As you can see we have two `displayxy()` methods. One takes two `int` primitive data type parameters and the other takes two `Integer` class types (these are reference variables) as parameters. Note that there is no difference in the way that we call the two versions of the method.

Here is the output of *Example 3*:

```
x is 10
y is 15
x * y = 150
```

```
xobj is 100
yobj is 150
xobj * yobj = 15000
```

```
xobj is 1000
yobj is 1500
xobj * yobj = 1500000
```

Note that autoboxing and auto-unboxing would work for passing `xobj` and `yobj` to the original `displayxy()` method. The point is we intentionally wanted a method that took `Integer` object parameters as well.

Value-returning Methods

After our discussion of void methods, value-returning methods should be a snap. One thing that should be mentioned at this point is not all methods have to print something although many of the examples do. You may recall that methods like `pow()`, `sqrt()` and `nextInt()` don't actually print *anything*, they simply return the requested information which usually gets stored in some variable.

The first step in creating value-returning methods is to turn the `void` into some other type. This type can be any primitive type or a class like `String`. Ultimately we would like to use our value-returning methods in some form of expression. The following example reiterates the ability of the `Math` methods to be used in such a way:

```
    cylvol = Math.PI * Math.pow(radius, 2.0) * height;
```

If you look at the `Math` class documentation, you will find a field called `PI` which is a named constant of the class. Recall that the definition of the `pow()` method from the `Math` class looks like this:

```
    public static double pow(double a, double b);
```

And we know that the purpose of `pow()` is to raise `a` to the `b` power. The key here is the return type of `double` to the left of the method name. Because this method returns a `double`, we can immediately use it in expressions. Void methods cannot be used this way.

Let us establish a moniker for identifying which is doing the calling and which is being called. The method doing the calling is the *calling method* which makes the method we are calling the *called method*.

One last item that needs mentioning is the use of the keyword `return`. As we will see, a `return` statement is what actually causes a value to be sent from the called method back to the calling method. If it helps, picture the two methods actually completing a full exchange of dialog. If the calling method passed parameters to the called method and the called method returned some value, then we could say that two methods are capable of carrying on a complete conversation. But, before we begin to give our methods personalities, let us consider *Example 4*.

In *Example 4*, three value-returning methods have been defined. They are `areaOfCircle()`, `circumference()` and `cylVolume()`. Before we dig too deeply, let us take a look at these methods in *Table 2* which describes our user-defined, value-returning methods much like we have done with all of the predefined methods thus far.

```

public class udmethoverload
{
    public static void displayxy(int xval, int yval)
    {
        System.out.println("x is " + xval);
        System.out.println("y is " + yval);
        System.out.println("x * y = " + xval * yval);
        System.out.println();
    }

    public static void displayxy(Integer xval, Integer yval)
    {
        System.out.println("xobj is " + xval);
        System.out.println("yobj is " + yval);
        System.out.println("xobj * yobj = " + xval * yval);
        System.out.println();
    }

    public static void main(String[] args)
    {
        int x, y;
        Integer xobj, yobj;

        x = 10;
        y = 15;
        displayxy(x, y);

        // JDK 5.0 autoboxing used here
        xobj = 100;
        yobj = 150;
        displayxy(xobj, yobj);

        xobj = 1000;
        yobj = 1500;
        displayxy(xobj, yobj);
    }
}

```

Example 3: Program that demonstrates method overloading.

User-defined, value-returning methods of <i>Example 4</i>
public static double areaOfCircle (double r) // Return the area of a circle as a double value using radius r.
public static double circumference (int d) // Return the circumference of a circle as a double using the integer diameter d.
public static double cylVolume (double r, double h) // Return the volume of a cylinder as a double using the cylinder radius r and height of h.

Table 2: Description of methods from Example 4.

At first glance the three methods may look similar. Although they share the same return type, that is the extent of their similarities. The `areaOfCircle()` method takes one `double` parameter, `circumference()` requires one integer primitive type parameter and `cylVolume()` requires two `double` parameters.

Now look closer at `cylVolume()`. Notice that this method makes use of another method! The `cylVolume()` method calls `areaOfCircle()`.

Here is a sample of the output with user-provided values in bold:

```

Enter the diameter of a cylinder: 5
Enter the height of the cylinder: 8
The radius is 2.500.
The diameter is 5.
The height is 8.
The circumference is 15.708.
The area of the base is 19.635.
The volume of the cylinder is 157.080.

```

```

import java.util.*;
public class vrmethod
{
    public static double areaOfCircle (double r)
    {
        return Math.PI * Math.pow(r, 2.0);
    }

    public static double circumference (int d)
    {
        return Math.PI * d;
    }

    public static double cylVolume (double r, double h)
    {
        return areaOfCircle(r) * h;
    }

    public static void main(String[] args)
    {
        Scanner kb = new Scanner(System.in);
        int diameter, height;
        double radius;
        double cir, area, volume;

        System.out.print("Enter the diameter of a cylinder: ");
        diameter = kb.nextInt();
        radius = diameter / 2.0;

        System.out.print("Enter the height of the cylinder: ");
        height = kb.nextInt();

        cir = circumference(diameter);
        area = areaOfCircle(radius);
        volume = cylVolume(radius, height);

        System.out.printf("The radius is %.3f.\n", radius);
        System.out.printf("The diameter is %d.\n", diameter);
        System.out.printf("The height is %d.\n", height);
        System.out.printf("The circumference is %.3f.\n", cir);
        System.out.printf("The area of the base is %.3f.\n", area);
        System.out.printf("The volume of the cylinder is %.3f.\n",
            volume);
    }
}

```

Example 4: Java program with three value-returning methods.

Parameters Revisited

As we discussed previously, we can have parameters of any primitive type. As demonstrated in *Example 3*, with method overloading we can also have reference variables as parameters. Remember that reference variables contain addresses of objects, not the objects themselves.

Keep these rules in mind when defining and using parameters:

- When passing primitive type variables as parameters, the formal parameter contains a copy of the data that is in the actual parameter. Nothing will happen to the data in the actual parameter.
- When passing reference variables as parameters, the formal parameter contains a copy of the address (a hash value) that is contained in the actual parameter. This means that changes made to the formal parameter are also changed for the actual parameter. This is because the actual and the formal are both pointing to the same object.

We need to be careful when dealing with `String` reference variables as parameters. Recall that simply using the assignment operator with a string is effectively creating a new instance of the `String`. Remember that for some string `s`:

```
s = "This string";
```

is the same as

```
s = new String("This string");
```

If your method assigns a new string to a formal parameter, the actual parameter will not be changed because `String` objects are *immutable*. *Example 5* demonstrates this phenomenon.

The output of *Example 5* is as follows:

```
s = String from main() method.  
str = String from alterstring() method!  
s = String from main() method.
```

As you can see, the value of `s` in the `main()` method did not change. The `StringBuffer` class can be used in these situations where you need to change the contents of the `String` object.

Keep in mind that you cannot use the assignment operator (`=`) to initialize your instantiated objects. *Example 6* demonstrates how to use `StringBuffer` to fix the problem in *Example 5*.

In *Example 6* we use the `replace()` method of the `StringBuffer` class and specify the beginning, end and new value of the string. By specifying zero as the first position and the current length of the string as the end position, we will effectively replace the entire string contents.

```
public class stringparm
{
    public static void alterstring(String str)
    {
        // Creates new instance. Does not replace
        // characters of existing object.
        str = "String from alterstring() method!";

        System.out.println("str = " + str);
    }

    public static void main(String[] args)
    {
        String s;

        s = "String from main() method.";
        System.out.println("s = " + s);

        alterstring(s);

        System.out.println("s = " + s);
    }
}
```

Example 5: Program demonstrating how a string assignment alters the formal parameter, but not the actual parameter.

```

public class stringBufferparm
{
    public static void alterstring(StringBuffer str)
    {
        // Replace characters of existing object.
        str.replace(0, str.length(),
                   "String from alterstring() method!");

        System.out.println("str = " + str);
    }

    public static void main(String[] args)
    {
        StringBuffer s;

        // Must use constructor
        s = new StringBuffer("String from main() method.");
        System.out.println("s = " + s);

        alterstring(s);

        System.out.println("s = " + s);
    }
}

```

Example 6: Program to use StringBuffer to fix the shortcomings of Example 5.