5.3: A closer look at methods and passing data to methods

IT1406 - Introduction to Programming

Level I - Semester 1

Overloading Methods

- In Java, it is possible to define two or more methods within the same class that share the same name, as long as their parameter declarations are different.
- When this is the case, the methods are said to be overloaded, and the process is referred to as *method overloading*.
- Method overloading is one of the ways that Java supports polymorphism.
- Here is a simple example that illustrates method overloading:

Overloading Methods

```
// Demonstrate method overloading.
class OverloadDemo {
	void test() {
		System.out.println("No parameters");
	}
	// Overload test for one integer parameter.
	void test(int a) {
		System.out.println("a: " + a);
	}
	// Overload test for two integer parameters.
	void test(int a, int b) {
		System.out.println("a and b: " + a + " " + b);
	}
	// Overload test for a double parameter
	double test(double a) {
		System.out.println("double a: " + a);
		return a*a;
	}
}
```

Overloading Methods

```
class Overload {
      public static void main(String args[]) {
            OverloadDemo ob = new OverloadDemo();
            double result;
            // call all versions of test()
            ob.test();
            ob.test(10);
            ob.test(10, 20);
            result = ob.test(123.25);
            System.out.println("Result of ob.test(123.25): " + result);
      }
}
```

- This program generates the following output:

```
No parameters
a: 10
a and b: 10 20
double a: 123.25
Result of ob.test(123.25): 15190.5625
```

Overloading Methods

- As you can see, this version of **OverloadDemo** does not define **test(int)**. Therefore, when **test( )** is called with an integer argument inside **Overload**, no matching method is found. However, Java can automatically convert an integer into a **double**, and this conversion can be used to resolve the call. Therefore, after **test(int)** is not found, Java elevates **i** to **double** and then calls **test(double)**.

Overloading Constructors

- In addition to overloading normal methods, you can also overload constructor methods. In fact, for most real-world classes that you create, overloaded constructors will be the norm, not the exception.

Overloading Constructors

Example:

```
class Box {
    double width;
    double height;
    double depth;
    // This is the constructor for Box.
    Box(double w, double h, double d) {
        width = w;
        height = h;
        depth = d;
    }
    // compute and return volume
    double volume() {
        return width * height * depth;
    }
}
```

Overloading Constructors

- As you can see, the **Box( )** constructor requires three parameters. This means that all declarations of **Box** objects must pass three arguments to the **Box( )** constructor. For example, the following statement is currently invalid:

```
Box ob = new Box();
```

- Since **Box( )** requires three arguments, it's an error to call it without them.
- Here is a program that contains an improved version of **Box** that does just that:

Overloading Constructors

```
class Box {
        double width;
        double height;
        double depth;
        // constructor used when all dimensions specified
        Box(double w, double h, double d) {
                width = w;
                height = h;
                depth = d;
        }
        // constructor used when no dimensions specified
        Box() {
                width = -1; // use -1 to indicate
                height = -1; // an uninitialized
                depth = -1; // box
        }
        // constructor used when cube is created
        Box(double len) {
                width = height = depth = len;
        }
```

Overloading Constructors

```
        // compute and return volume
        double volume() {
                return width * height * depth;
        }
}
class OverloadCons {
        public static void main(String args[]) {
        // create boxes using the various constructors
                Box mybox1 = new Box(10, 20, 15);
                Box mybox2 = new Box();
                Box mycube = new Box(7);
                double vol;
                vol = mybox1.volume();
                System.out.println("Volume of mybox1 is " + vol);
                vol = mybox2.volume();
                System.out.println("Volume of mybox2 is " + vol);
                vol = mycube.volume();
                System.out.println("Volume of mycube is " + vol);
        }
}
```

Overloading Constructors

- The output produced by this program is shown here:

  Volume of mybox1 is 3000.0
  Volume of mybox2 is -1.0
  Volume of mycube is 343.0

- As you can see, the proper overloaded constructor is called based upon the parameters specified when **new** is executed.

Using Objects as Parameters

- So far, we have only been using simple types as parameters to methods.
- However, it is both correct and common to pass objects to methods.
- For example, consider the following short program:

```
// Objects may be passed to methods.
class Test {
        int a, b;
        Test(int i, int j) {
                a = i;
                b = j;
}
```

Using Objects as Parameters

```
        // return true if o is equal to the invoking object
        boolean equalTo(Test o) {
                if(o.a == a && o.b == b) return true;
                else return false;
        }
}
class PassOb {
        public static void main(String args[]) {
                Test ob1 = new Test(100, 22);
                Test ob2 = new Test(100, 22);
                Test ob3 = new Test(-1, -1);
                System.out.println("ob1 == ob2: " + ob1.equalTo(ob2));
                System.out.println("ob1 == ob3: " + ob1.equalTo(ob3));
        }
        }
```

Using Objects as Parameters

- This program generates the following output:

```
ob1 == ob2: true
ob1 == ob3: false
```

- As you can see, the **equalTo( )** method inside **Test** compares two objects for equality and returns the result. That is, it compares the invoking object with the one that it is passed.
- If they contain the same values, then the method returns **true**. Otherwise, it returns **false**.
- Notice that the parameter **o** in **equalTo( )** specifies **Test** as its type.

A Closer Look at Argument Passing

There are two ways that a computer language can pass an argument to a subroutine.

1. ***call-by-value***

   This approach copies the *value* of an argument into the formal parameter of the subroutine. Therefore, changes made to the parameter of the subroutine have no effect on the argument.

2. ***call-by-reference***

   In this approach, a reference to an argument (not the value of the argument) is passed to the parameter. Inside the subroutine, this reference is used to access the actual argument specified in the call. This means that changes made to the parameter will affect the argument used to call the subroutine.

A Closer Look at Argument Passing

- As you will see, although Java uses call-by-value to pass all arguments, the precise effect differs between whether a primitive type or a reference type is passed.
- When you pass a primitive type to a method, it is passed by value. Thus, a copy of the argument is made, and what occurs to the parameter that receives the argument has no effect outside the method.
- For example, consider the following program:

A Closer Look at Argument Passing

```
// Primitive types are passed by value.
class Test {
      void meth(int i, int j) {
            i *= 2;
            j /= 2;
      }
}
class CallByValue {
      public static void main(String args[]) {
            Test ob = new Test();
            int a = 15, b = 20;
            System.out.println("a and b before call: " + a + " " + b);
            ob.meth(a, b);
            System.out.println("a and b after call: " + a + " " + b);
      }
}
```

A Closer Look at Argument Passing

- The output from this program is shown here:

  a and b before call: 15 20
  a and b after call: 15 20

- As you can see, the operations that occur inside **meth( )** have no effect on the values of **a** and **b** used in the call; their values here did not change to 30 and 10.
- When you pass an object to a method, the situation changes dramatically, because objects are passed by what is effectively call-by-reference.
- When you create a variable of a class type, you are only creating a reference to an object.
- Thus, when you pass this reference to a method, the parameter that receives it will refer to the same object as that referred to by the argument. This effectively means that objects act as if they are passed to methods by use of call-by-reference.
- Changes to the object inside the method *do* affect the object used as an argument.

A Closer Look at Argument Passing

```
// Objects are passed through their references.
class Test {
        int a, b;
        Test(int i, int j) {
                a = i;
                b = j;
        }
        // pass an object
        void meth(Test o) {
                o.a *= 2;
                o.b /= 2;
        }
}
class PassObjRef {
        public static void main(String args[]) {
                Test ob = new Test(15, 20);
                System.out.println("ob.a and ob.b before call: " + ob.a + " " + ob.b);
                ob.meth(ob);
                System.out.println("ob.a and ob.b after call: " + ob.a + " " + ob.b);
        }
}
```

A Closer Look at Argument Passing

- This program generates the following output:

  ob.a and ob.b before call: 15 20
  ob.a and ob.b after call: 30 10

- As you can see, in this case, the actions inside **meth( )** have affected the object used as an argument.
- When an object reference is passed to a method, the reference itself is passed by use of call-by-value. However, since the value being passed refers to an object, the copy of that value will still refer to the same object that its corresponding argument does.

Returning Objects

- A method can return any type of data, including class types that you create. For example, in the following program, the **incrByTen( )** method returns an object in which the value of **a** is sten greater than it is in the invoking object.

Returning Objects

```
// Returning an object.
class Test {
        int a;
        Test(int i) {
                a = i;
        }
        Test incrByTen() {
                Test temp = new Test(a+10);
                return temp;
        }
}
class RetOb {
        public static void main(String args[]) {
                Test ob1 = new Test(2);
                Test ob2;
                ob2 = ob1.incrByTen();
                System.out.println("ob1.a: " + ob1.a);
                System.out.println("ob2.a: " + ob2.a);
                ob2 = ob2.incrByTen();
                System.out.println("ob2.a after second increase: " + ob2.a);
        }
}
```

Returning Objects

- The output generated by this program is shown here:

```
ob1.a: 2
ob2.a: 12
ob2.a after second increase: 22
```

- As you can see, each time **incrByTen( )** is invoked, a new object is created, and a reference to it is returned to the calling routine.
- The preceding program makes another important point: Since all objects are dynamically allocated using **new**, you don't need to worry about an object going out-of-scope because the method in which it was created terminates. The object will continue to exist as long as there is a reference to it somewhere in your program. When there are no references to it, the object will be reclaimed the next time garbage collection takes place.