A CLOSE LOOK AT METHODS AND CLASSES

Asmaliza Ahzan
IVERSON ASSOCIATES SDN BHD

A Close Look at Methods and Classes

Controlling Access to Class Members

- Access level modifiers determine whether other classes can use a particular field or invoke a particular method.
- There are two levels of access control:
    - At the top level - public, or package-private (no explicit modifier).
    - At the member level - public, private, protected, or package-private (no explicit modifier).
- A class may be declared with the modifier public, in which case that class is visible to all classes everywhere.
- If a class has no modifier (the default, also known as package-private), it is visible only within its own package.
- At the member level, you can also use the public modifier or no modifier (package-private) just as with top-level classes, and with the same meaning.
- For members, there are two additional access modifiers: private and protected.
- The private modifier specifies that the member can only be accessed in its own class.
- The protected modifier specifies that the member can only be accessed within its own package (as with package-private) and, in addition, by a subclass of its class in another package.

Modifier	Class	Package	Subclass	World
public	Y	Y	Y	Y
protected	Y	Y	Y	N
no modifier	Y	Y	N	N
private	Y	N	N	N

Tips on Choosing an Access Level

- If other programmers use your class, you want to ensure that errors from misuse cannot happen.
- Access levels can help you do this.
    - Use the most restrictive access level that makes sense for a particular member. Use private unless you have a good reason not to.
    - Avoid public fields except for constants.

Encapsulation

- Encapsulation in Java is a mechanism of wrapping the data (variables) and code acting on the data (methods) together as a single unit.
- In encapsulation, the variables of a class will be hidden from other classes and can be accessed only through the methods of their current class.
- Therefore, it is also known as data hiding.

To achieve encapsulation in Java

- Declare the variables of a class as private.
- Provide public setter and getter methods to modify and view the variables values.

```
public class EncapTest {
   private String name;
   private String idNum;
   private int age;

   public int getAge() {
      return age;
   }

   public String getName() {
      return name;
   }

   public String getIdNum() {
      return idNum;
   }

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

   public void setName(String newName) {
      name = newName;
   }

   public void setIdNum( String newId) {
      idNum = newId;
   }
}
```

- The public setXXX() and getXXX() methods are the access points of the instance variables of the EncapTest class.
- Normally, these methods are referred as getters and setters.
- Therefore, any class that wants to access the variables should access them through these getters and setters.

Pass Objects to Methods

- Java is strictly pass-by-value.
- Object references can be parameters.
- Call by value is used, but now the value is an object reference.
- This reference can be used to access the object and possibly change it.

```
public void moveCircle(Circle circle, int deltaX, int deltaY) {
    // code to move origin of circle to x+deltaX, y+deltaY
    circle.setX(circle.getX() + deltaX);
    circle.setY(circle.getY() + deltaY);

    // code to assign a new reference to circle
    circle = new Circle(0, 0);
}
```

Let the method be invoked with these arguments:

```
moveCircle(myCircle, 23, 56)
```

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 ten greater than it is in the invoking object.

```
// 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);
	}
}
```

Method Overloading

- 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.
- When an overloaded method is invoked, Java uses the type and/or number of arguments as its guide to determine which version of the overloaded method to actually call.
- Thus, overloaded methods must differ in the type and/or number of their parameters.

```
// 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;
	}
}

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);
	}
}
```

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.

```
/* Here, Box defines three constructors to initialize
the dimensions of a box various ways.
*/
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;
	}

	// compute and return volume
	double volume() {
		return width * height * depth;
	}
}
```

Recursion

- Java supports recursion. Recursion is the process of defining something in terms of itself.
- As it relates to Java programming, recursion is the attribute that allows a method to call itself.
- A method that calls itself is said to be recursive.

The classic example of recursion is the computation of the factorial of a number. The factorial of a number N is the product of all the whole numbers between 1 and N. For example, 3 factorial is 1 × 2 × 3 ×, or 6. Here is how a factorial can be computed by use of a recursive method.

```
// A simple example of recursion.
class Factorial {
// this is a recursive method
	int fact(int n) {
		int result;
		if (n == 1)
			return 1;
		result = fact(n - 1) * n;
		return result;
	}
}

class Recursion {
	public static void main(String args[]) {
		Factorial f = new Factorial();
		System.out.println("Factorial of 3 is " + f.fact(3));
		System.out.println("Factorial of 4 is " + f.fact(4));
		System.out.println("Factorial of 5 is " + f.fact(5));
	}
}
```

Understanding static

- There will be times when you will want to define a class member that will be used independently of any object of that class.
- Normally, a class member must be accessed only in conjunction with an object of its class.
- However, it is possible to create a member that can be used by itself, without reference to a specific instance.
- To create such a member, precede its declaration with the keyword static.
- When a member is declared static, it can be accessed before any objects of its class are created, and without reference to any object.
- You can declare both methods and variables to be static.
- The most common example of a static member is main( ). main( ) is declared as static because it must be called before any objects exist.
- Instance variables declared as static are, essentially, global variables.
- When objects of its class are declared, no copy of a static variable is made.
- Instead, all instances of the class share the same static variable.

Methods declared as static have several restrictions.

- They can only directly call other static methods.
- They can only directly access static data.
- They cannot refer to this or super in any way.

```
// Demonstrate static variables, methods, and blocks.
class UseStatic {
	static int a = 3;
	static int b;

	static void meth(int x) {
		System.out.println("x = " + x);
		System.out.println("a = " + a);
		System.out.println("b = " + b);
	}

	static {
		System.out.println("Static block initialized.");
		b = a * 4;
	}

	public static void main(String args[]) {
		meth(42);
	}
}
```

Introducing Nested and Inner Classes

- It is possible to define a class within another class; such classes are known as nested classes.
- The scope of a nested class is bounded by the scope of its enclosing class. Thus, if class B is defined within class A, then B does not exist independently of A.
- A nested class has access to the members, including private members, of the class in which it is nested. However, the enclosing class does not have access to the members of the nested class.
- A nested class that is declared directly within its enclosing class scope is a member of its enclosing class.
- It is also possible to declare a nested class that is local to a block.

There are two types of nested classes: static and non-static.

- A static nested class is one that has the static modifier applied.
- Because it is static, it must access the non-static members of its enclosing class through an object.
- That is, it cannot refer to non-static members of its enclosing class directly.
- Because of this restriction, static nested classes are seldom used.

The most important type of nested class is the inner class. An inner class is a non-static nested class. It has access to all of the variables and methods of its outer class and may refer to them directly in the same way that other non-static members of the outer class do.

```
// Demonstrate an inner class.
class Outer {
	int outer_x = 100;

	void test() {
		Inner inner = new Inner();
		inner.display();
	}

// this is an inner class
	class Inner {
		void display() {
			System.out.println("display: outer_x = " + outer_x);
		}
	}
}

class InnerClassDemo {
	public static void main(String args[]) {
		Outer outer = new Outer();
		outer.test();
	}
}
```

Varargs: Variable-Length Arguments

- Beginning with JDK 5, Java has included a feature that simplifies the creation of methods that need to take a variable number of arguments.
- This feature is called varargs and it is short for variable-length arguments.
- A method that takes a variable number of arguments is called a variable-arity method, or simply a varargs method.

```
// Use an array to pass a variable number of
// arguments to a method. This is the old-style
// approach to variable-length arguments.
class PassArray {
   static void vaTest(int v[]) {
          System.out.print("Number of args: " + v.length + " Contents: ");
          for (int x : v)
                System.out.print(x + " ");
          System.out.println();
   }

   public static void main(String args[]) {
          // Notice how an array must be created to
          // hold the arguments.
          int n1[] = { 10 };
          int n2[] = { 1, 2, 3 };
          int n3[] = {};
          vaTest(n1); // 1 arg
          vaTest(n2); // 3 args
          vaTest(n3); // no args
   }
}
```