Programming in Java

Lecture 07: A Closer Look at Methods and Classes(2)

Mahesh Kumar
Assistant Professor (Adhoc)

Department of Computer Science
Acharya Narendra Dev College
University of Delhi

Course webpage
[http://www.mkbhandari.com/mkwiki]

Outline

1. Recursion
2. Introducing Access Control
3. Understanding static
4. Introducing final

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.

Recursion

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

Q What will be the output?

Factorial of 3 is 6
Factorial of 4 is 24
Factorial of 5 is 120

Recursion

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

1 *fact(3) = fact(2) * 3*

2 *fact(2) = fact(1) * 2*

3 *fact(1) = 1*

Decomposition

Recursion

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

Decomposition

1 *fact(3) = fact(2) * 3*

2 *fact(2) = fact(1) * 2*

3 *fact(1) = 1*

2 *fact(2) = 1 * 2*

1 *fact(3) = 2 * 3*

Backtracking

Recursion

- When a method calls itself, new local variables and parameters are allocated storage on the stack, and the method code is executed with these new variables from the start
- As each recursive call returns, the old local variables and parameters are removed from the stack, and execution resumes at the point of the call inside the method.
- Recursive versions of many routines may execute a bit more slowly than the iterative equivalent because of the added overhead of the additional method calls.
- Many recursive calls to a method could cause a stack overrun. Because storage for parameters and local variables is on the stack and each new call creates a new copy of these variables, it is possible that the stack could be exhausted.
- If this occurs, the Java run-time system will cause an exception.

Recursion

- The main advantage to recursive methods is that they can be used to create clearer and simpler versions of several algorithms than can their iterative relatives.
- For example: Quick Sort Algorithm, some types of AI-related algorithms
- When writing recursive methods, you must have an if statement somewhere to force the method to return without the recursive call being executed(**base condition**). If you don’t do this, once you call the method, it will never return.

Recursion

```
// Another example that uses recursion.
class RecTest {
	int values[ ];
	RecTest(int i) {
		values = new int[i];
	}
	// display array -- recursively
	void printArray(int i) {
		if(i==0) return;
		else printArray(i-1);
		System.out.println("[" + (i-1) + "] " + values[i-1]);
	}
}
class Recursion2 {
	public static void main(String args[ ]) {
		RecTest ob = new RecTest(10);
		for(int i=0; i<10; i++) {
			ob.values[i] = i;
		}
		ob.printArray(10);
	}
}
```

Recursion

```
// Another example that uses recursion.
class RecTest {
	int values[ ];
	RecTest(int i) {
		values = new int[i];
	}
	// display array -- recursively
	void printArray(int i) {
		if(i==0) return;
		else printArray(i-1);
		System.out.println("[" + (i-1) + "] " + values[i-1]);
	}
}
class Recursion2 {
	public static void main(String args[ ]) {
		RecTest ob = new RecTest(10);
		for(int i=0; i<10; i++) {
			ob.values[i] = i;
		}
		ob.printArray(10);
	}
}
```

Q What will be the output?

```
[0] 0
[1] 1
[2] 2
[3] 3
[4] 4
[5] 5
[6] 6
[7] 7
[8] 8
[9] 9
```

Introducing Access Control

- Access/Visibility specifiers/modifiers/control are the mechanism by which you can precisely control access to the various members of a class.
- How a member can be accessed is determined by the access modifier attached to its declaration.
- Java supplies a rich set of access modifiers. Some aspects of access control are related mostly to inheritance or packages.
- Java's access modifiers are:

  1. *public* - accessible from every where
  2. *private - within the class*
  3. *protected* - applies only when inheritance is involved
  4. *default – only within the same package.*

Introducing Access Control

	default	private	protected	public
Same Class	Yes	Yes	Yes	Yes
Same package subclass	Yes	No	Yes	Yes
Same package non-subclass	Yes	No	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

Introducing Access Control

```
// This program demonstrates the difference between public and private.

class Test {
	int a;			// default access
	public int b;		// public access
	private int c;		// private access

	// methods to access c
	void setc(int i) { // set c's value
		c = i;
	}

	int getc( ) { // get c's value
		return c;
	}
}
```

```
class AccessTest {
	public static void main(String args[ ]) {
		Test ob = new Test();
		// These are OK, a and b may be accessed directly
		ob.a = 10;
		ob.b = 20;

		// This is not OK and will cause an error
		ob.c = 100;		// Error

		// You must access c through its methods
		ob.setc(100);		// OK

		System.out.println("a, b, and c: " + ob.a + " " +
		ob.b + " " + ob.getc( ) );
	}
}
```

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.

Understanding 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.

Understanding Static

- Methods declared as static have several restrictions:
  1. They can only directly call other static methods.
  2. They can only directly access static data.
  3. They cannot refer to this or super in any way.

- If you need to do computation in order to initialize your static variables, you can declare a static block that gets executed exactly once, when the class is first loaded.

Understanding Static

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

1. As soon as the UseStatic class is loaded, all of the static statements are run.
2. First, a is set to 3, then the static block executes, which prints a message and then initializes b to a*4 or 12.
3. Then main( ) is called, which calls meth( ), passing 42 to x.
4. The three println( ) statements refer to the two static variables a and b, as well as to the local variable x.

Q What will be the output?

Understanding Static

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

1. As soon as the UseStatic class is loaded, all of the static statements are run.
2. First, a is set to 3, then the static block executes, which prints a message and then initializes b to a*4 or 12.
3. Then main( ) is called, which calls meth( ), passing 42 to x.
4. The three println( ) statements refer to the two static variables a and b, as well as to the local variable x.

Q What will be the output?

```
Static block initialized.
x = 42
a = 3
b = 12
```

Understanding Static

```
// Accessing Static members outside the class

class StaticDemo {
        static int a = 42;
        static int b = 99;

        static void callme( ) {
                System.out.println("a = " + a);
        }
}

class StaticByName {
        public static void main(String args[ ]) {
                StaticDemo.callme( );
                System.out.println("b = " + StaticDemo.b);
        }
}
```

Q What will be the output?

Understanding Static

```
// Accessing Static members outside the class

class StaticDemo {
	static int a = 42;
	static int b = 99;

	static void callme( ) {
		System.out.println("a = " + a);
	}
}

class StaticByName {
	public static void main(String args[ ]) {
		StaticDemo.callme( );
		System.out.println("b = " + StaticDemo.b);
	}
}
```

Q What will be the output?

```
a = 42
b = 99
```

Introducing Final

- A variable can be declared as final.
- Doing so prevents its contents from being modified, making it, essentially, a constant.
- We must initialize a final variable when it is declared. You can do this in one of two ways:

1. You can give it a value when it is declared (commonly used).

```
final int FILE_NEW  = 1;            //1.  Can be used as constant in the subsequent parts of your program
final int MAX_MARKS  = 100;     // 2. Common coding convention to use all uppercase identifiers for final fields
final int SPEEDLIMIT=60;
```

2. You can assign it a value within a constructor (blank final variable).

Introducing Final

- In addition to fields, both method parameters and local variables can be declared final.
- Declaring a parameter final prevents it from being changed within the method.
- Declaring a local variable final prevents it from being assigned a value more than once.
- The keyword final can also be applied to methods, but its meaning is substantially different than when it is applied to variables. (Will be discussed in Inheritance)

References

Reference for this topic

- [Book: Java: The Complete Reference, Ninth Edition: Herbert Schildt]
  https://www.amazon.in/Java-Complete-Reference-Herbert-Schildt/dp/0071808558

- [Web: GeeksforGeeks]
  https://www.geeksforgeeks.org/java/

- [Web: Java T Point tutorial]
  https://www.javatpoint.com/java-tutorial