Java
Chapter 7: A Closer Look at Methods and Classes

Slides material compiled from
Java - The Complete Reference 9th Edition
By
Herbert Schildt

Overloading Methods

- **Method Overloading**
  - two or more methods within the same class that share the same name, but their parameter declarations (type and/or number) are different
  - one of the ways that Java supports **polymorphism**
  - Java uses the type and/or number of arguments to determine which version of the overloaded method to actually call
  - ***return type alone is insufficient to distinguish two versions of a method***

In some cases, Java's automatic type conversions can play a role in overload resolution

Also possible to overload constructors

Example (OverLoadDemo.java)

```
class OverloadDemo {

void test() {
   System.out.println("No parameters");
}
void test(int a, int b) {
   System.out.println("a and b: " + a + " " +
b);
}
double test(double a) {
   System.out.println("double a: " + a);
   return a*a;
}
}
```

Example (Overload.java)

```
class Overload {
public static void main(String args[]) {
  OverloadDemo ob = new OverloadDemo();
  int i = 88;
  double result;

  ob.test();
  ob.test(10, 20);
  result ob.test(123.25); // this will invoke test(double)
  System.out.println("Result: " + result);
  result = ob.test(i); // this will invoke test(double)
  System.out.println("Result: " + result);

}
```

Using Objects as Parameters

```
class Test {
  int a, b;
  Test(int i, int j) {
     a = i; b = j;
   }
   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

```
class Test {
  int a, b;
  Test(int i, int j) {
     a = i; b = j;
   }
   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));
  }
}
```

OUTPUT
ob1 == ob2: true
ob1 == ob3: false

A Closer Look at Argument Passing

	CALL BY VALUE	CALL BY REFERENCE
1.	The value of the argument is copied into the formal parameter of the method.	A reference to an argument (not the value of the argument) is passed to the parameter. Inside the method, this reference is used to access the actual argument specified in the call.
2.	Changes made to the parameter will not affect the argument used to call the method.	Changes made to the parameter will affect the argument used to call the method.
3.	Simple types (int, float, char, etc) are passed as call by value	Objects are passed as call by reference.

Call by Value

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

Call by Value

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

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

Call by Reference

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

Call by Reference

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

OUTPUT
Before call: 15 20
After call: 30 10

Returning Objects

```
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: " + ob2.a);
   }
}
```

Returning Objects

```
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: " + ob2.a);
   }
}
```

OUTPUT
ob1.a: 2
ob2.a: 12
ob2.a : 22

Recursion

- Recursion is the attribute that allows a method to call itself

```
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));
   }
}
```

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));
   }
}
```

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

Recursion (Another Example)

```
// display array -- recursively
class RecTest {
   int values[];
   RecTest(int i) {  values = new int[i]; }
   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);
       int i;
       for(i=0; i<10; i++) ob.values[i] = i;
       ob.printArray(10);
  }
}
```

Recursion (Another Example)

```
// display array -- recursively
class RecTest {
   int values[];
   RecTest(int i) {  values = new int[i]; }
   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);
       int i;
       for(i=0; i<10; i++) ob.values[i] = i;
       ob.printArray(10);
  }
}
```

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 specifiers in Java provide a means for containing the name space and scope of variables and methods.

private	can be accessed only within the class and not outside it
no modifier (default)	can be accessed outside the class in which they are declared but only by classes (including subclasses) within the same package
protected	can be accessed outside the class in which they are declared within the same package but only by subclasses outside the package
public	can be accessed anywhere

	Private	No Modifier	Protected	Public
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

Understanding **static**

- **Static method/instance variable:** Can be accessed before any objects of its class are created, and without reference to any object
- **main( )** is declared as **static** because it must be called before any objects exist
- **static** instance variables
  - Are essentially global variables because
  - all instances of the class share the same **static** variable
- Methods declared as **static** have several restrictions:
  - Can only directly call other **static** methods
  - Can only directly access **static** data
  - Cannot refer to **this** or **super** in any way

Initializing static variables

- declare a **static** block that gets executed exactly once, when the class is first loaded

```
class UseStatic {
        static int a = 3;
        static int b;
        static void meth(int x) {
                System.out.println("x = " + x +" a = " + a +
                        " b = " + b);
        }
        static {
            b = a * 4;
         }
        public static void main(String args[]) {
                meth(42);
        }
}
```

OUTPUT
x = 42
a = 3
b = 12

Accessing static members

- **static** methods and variables can be used independently of any object; specify the name of their class followed by the dot operator

```
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);
    }
}
```

Accessing static members

```
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);
    }
}
```

OUTPUT
a = 42
b = 99

Introducing **final**

- Declaring a member **final** prevents its contents from being modified
  - Must be initialized when it is declared
  - Or assign it a value within a constructor
- **Final data member**: contents cannot be modified
  - A final variable has to be initialized at the time of declaration.
  - `final int YES = 1; final int NO = 2;`
  - Variables declared as final do not occupy memory on a per instance basis; essentially named constants of the class
- **Final method member**: cannot be overridden by subclasses

Nested and Inner Classes

- **Nested class**: a class defined within another class
  - If class B is declared inside class A, then B is known to A but not outside A.
  - Members of A can be accessed by B, but A does not have access to members of B.
- Nested classes can be static or non static.
  - Non static Nested classes are known as **Inner Class**
  - Inner class has access to all the members of its outer class (including private members)
  - outer class can access members of an Inner Class by creating an object of the Inner Class

Using Command-Line Arguments

- A command-line argument is the information that directly follows the program's name on the command line when it is executed
- stored as strings in a **String** array passed to the **args** parameter of **main( )**

```
// Display all command-line arguments.
class CommandLine {
public static void main(String args[]) {
  for(int i=0; i<args.length; i++)
     System.out.println("args["+i+"]: " + args[i]);
}
}
```

Note: All command-line arguments are passed as strings. Convert numeric values to their internal forms manually

Varargs: Variable-Length Arguments

- ***varargs method:*** takes a variable number of argument which is specified by three periods (**…**)

```
class VarArgs {
static void vaTest(int ... v) {
   System.out.print("No of args: " + v.length +"\n Contents: ");
  for(int x : v)
     System.out.print(x + " ");
   System.out.println();
  }
public static void main(String args[]){
    vaTest(10); // 1 arg
    vaTest(1, 2, 3); // 3 args
    vaTest(); // no args
}
}
```

  - **vaTest( )** can be called with zero or more arguments
  - **v** is implicitly declared as an array of type **int[ ]**
  - Inside **vaTest( )**, **v** is accessed using the normal array syntax

Varargs: Variable-Length Arguments

```
class VarArgs {
static void vaTest(int ... v) {
   System.out.print("No. of args: " + v.length +
                    "Contents: ");
  for(int x : v)
     System.out.print(x + " ");
   System.out.println();
  }
public static void main(String args[]){
    vaTest(10); // 1 arg
    vaTest(1, 2, 3); // 3 args
    vaTest(); // no args
}
}
```

OUTPUT

No. of args: 1 Contents: 10
No. of args: 3 Contents: 1 2 3
No. of args: 0 Contents:

Varargs with standard arguments

- Can have “normal” parameters along with a variable-length parameter
  - the variable-length parameter must be the last parameter
  - there can be only one varargs parameter

```
class VarArgs2 {
  static void vaTest(String msg, int ... v) {
     System.out.print(msg + v.length + " Contents: ");
     for(int x : v)
        System.out.print(x + " ");
     System.out.println();
  }
  public static void main(String args[])
  {
     vaTest("One vararg: ", 10);
     vaTest("Three varargs: ", 1, 2, 3);
     vaTest("No varargs: ");
  }
}
```

Varargs methods with standard arguments

```
class VarArgs2 {
  static void vaTest(String msg, int ... v) {
     System.out.print(msg + v.length + " Contents: ");
     for(int x : v)
        System.out.print(x + " ");
     System.out.println();
  }
  public static void main(String args[])
  {
     vaTest("One vararg: ", 10);
     vaTest("Three varargs: ", 1, 2, 3);
     vaTest("No varargs: ");
  }
}
```

OUTPUT

One vararg: 1 Contents: 10
Three varargs: 3 Contents: 1 2 3
No varargs: 0 Contents:

Overloading Vararg Methods

```
class VarArgs3 {
  static void vaTest(int ... v) {
     System.out.print("vaTest(int ...): " +
     "Number of args: " + v.length + " Contents: ");
     for(int x : v)
          System.out.print(x + " ");
     System.out.println();
  }
  static void vaTest(boolean ... v) {
    System.out.print("vaTest(boolean ...) " +
    "Number of args: " + v.length + " Contents: ");
    for(boolean x : v)
       System.out.print(x + " ");
    System.out.println();
  }
```

Overloading Vararg Methods

```
static void vaTest(String msg, int ... v) {
   System.out.print("vaTest(String, int ...): "
   + msg + v.length + " Contents: ");
   for(int x : v)
      System.out.print(x + " ");
   System.out.println();
}
public static void main(String args[])
{
        vaTest(1, 2, 3);
        vaTest("Testing: ", 10, 20);
        vaTest(true, false, false);
}
}
```

Overloading Vararg Methods

OUTPUT
vaTest(int ...): Number of args: 3 Contents: 1 2 3
vaTest(String, int ...): Testing: 2 Contents: 10 20
vaTest(boolean ...) Number of args: 3 Contents: true false false

- **NOTE** A varargs method can also be overloaded by a non-varargs method
  - For example, vaTest(int x) is a valid overload of vaTest( )
  - This version is invoked only when one int argument is present
  - When two or more int arguments are passed, the varargs version vaTest (int…v) is used