Understanding Method Binding in Java

By Kishori Sharan

Published on: February 6, 2012 at www.jdojo.com

Method binding is a mechanism of associating a method call in Java code to the declaration and implementation of the method being called. When we mention the phrase “declaration of the method”, we are referring to the method signature, which consists of the method name, and the order and the data type of its parameters. When we mention the phrase “implementation of the method”, we are referring to the body of the method that executes at runtime as part of the method call.

The compiler always decides the signature of the called method. Depending on the method type (static, non-static, and interface), the compiler or the runtime decides what implementation of the method is executed at runtime.

Let us consider three Java programs listed in Listing 1, Listing 2, and Listing 3.

Listing 1: An Employee class

```java
// Employee.java
package com.jdojo.blogs.methodbinding;

public class Employee {
    public void setSalary(double salary) {
        System.out.println("Inside Employee.setSalary()”);
    }
}
```

Listing 2: A Manager class

```java
// Manager.java
package com.jdojo.blogs.methodbinding;

public class Manager extends Employee {
    public void setSalary(int salary) {
        System.out.println("Inside Manager.setSalary()”);
    }
}
```

Listing 3: A Test class to test method binding

```java
// Test.java
package com.jdojo.blogs.methodbinding;

public class Test {
    public static void main(String[] args) {
        /* Create a Manager object and assign its reference to a variable emp of the Employee type */
        Employee emp = new Manager();

        /* Call the setSalary() method */
    }
}
int salary = 12000;
emp.setSalary(salary); /* Which setSalary() method is called.
Employee.setSalary() or Manager.setSalary() */
}

Output:

Can you tell what is printed when the Test class is run?

The following is my favorite question, which I ask in written tests for Java developers. So far, none of the Java developers has answered this question correctly. Let us see if you can answer this question correctly with a correct explanation.

What is printed on the standard output when the Test class is run? Does it print “Inside Employee.setSalary()” or “Inside Manager.setSalary()”?

The correct answer is “Inside Employee.setSalary()”. To understand the output, you need to read the full post and make sure that you understand the method binding mechanism in Java.

The following statement in the Test class declares an emp reference variable of Employee type and assign it an object of the Manager class.

Employee emp = new Manager();

The above statement compiles and runs fine, because of the upcasting rules in Java, which states, “An object of a subclass can be assigned to a variable of a superclass.”

The following snippet of code calls the setSalary() method on the emp reference variable.

int salary = 12000;
emp.setSalary(salary);

The setSalary() method is a non-static method. The emp reference variable holds a reference to a Manager object. A common understanding (and misunderstanding as well) among the Java developers is that a non-static method call is bound at runtime, so the Java runtime will decide which setSalary() method to call in the emp.setSalary(salary) statement. However, that is only half of the story. Since most of the textbooks go only up to this extent to describe the method binding mechanism, readers of those textbooks never get past this fact, and hence, keep making mistakes like answering the above question wrong. Most of the Java developers think that the emp.setSalary(salary) method call is bound to the setSalary(int) method of the Manager class, because the emp reference variable holds a reference of a Manager object and the method call uses an int argument, which is matched to the setSalary() method of the Manager class. If you thought the same way (and most likely you did), you are wrong. So, here is the full story of method binding in Java.

The following are the two parts of a method, and two steps in a method binding.

- Method signature - the method name, the order and the data type of the formal parameters
- Method implementation - body of the method, which executes at runtime
- Compile-time binding - performed by the compiler
- Runtime binding - performed by the Java runtime
Table 1 lists the method types, method parts and the stages when they are bound.

Table 1: Method types and the stages when a part of the method is bound

<table>
<thead>
<tr>
<th>Method Type</th>
<th>Compile-time Binding</th>
<th>Runtime Binding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-Static</td>
<td>Method Signature</td>
<td>Method Implementation</td>
</tr>
<tr>
<td>Static</td>
<td>Method Signature</td>
<td>Method Implementation</td>
</tr>
</tbody>
</table>

It is clear from Table 1 that the signature of a method is always bound at compile-time. That is, the compiler always decides the signature of the called method - it does not matter if a method is a static or non-static method. Let us apply this rule to the following statement.

emp.setSalary(salary);

First, the compiler determines the compile-time (also known as static or declared type) of the emp reference variable. Since the emp reference variable has been declares of Employee type, the compile-time type of the emp variable is Employee. The compiler looks for a compatible method signature (declared or inherited), setSalary(int), in the Employee class. It finds a setSalary(double) method in the Employee class, which is compatible to setSalary(int) method call, after the argument widening from int to double. The compiler binds the emp.setSalary(salary) method call to a method setSalary(double). Note that the compiler has bound the call to a method setSalary(), which accepts a double argument. The argument of the called method setSalary() has already been decided to double, and it cannot be changed at runtime. Can we prove this fact? Yes. We can prove it by dissembling the class file for the Test class. We need to use the javap tool to dissemble the class file. The javap tool is located in the JAVA_HOME\bin folder, where Java_HOME is the JDK installation folder. You can run the javap tool as follows. The -c option dissembles the byte code and prints it on the standard output.

javap -cp YOUR-CLASSPATH-GOES-HERE -c com.jdojo.blogs.methodbinding.Test

Output of the javap tool:

Compiled from "Test.java"
public class com.jdojo.blogs.methodbinding.Test {
    public com.jdojo.blogs.methodbinding.Test();
    Code:
    0: aload_0
    1: invokespecial #1 // Method java/lang/Object."<init>"():V
    4: return

    public static void main(java.lang.String[]);
    Code:
    0: new           #2 // class com/jdojo/blogs/methodbinding/Manager
    3: dup
    4: invokespecial #3 // Method com/jdojo/blogs/methodbinding/Manager."<init>"():V
    7: astore_1
    8: sipush        12000
    11: istore_2
    12: aload_1
    13: iload_2
    14: i2d
    15: invokevirtual #4 // Method com/jdojo/blogs/methodbinding/Employee.setSalary:(D)V
We are interested only in the following line in the disassembled code.

15: invokevirtual #4 // Method com/jdojo/blogs/methodbinding/Employee.setSalary:(D)V

This is the byte code instruction that is generated for the statement `emp.setSalary(salary)` in the `Test` class. The last part `setSalary(D)V` means a `setSalary` method, which accepts a `double` (`D` for `double`) parameter and returns `void` (`V` for `void`). The instruction `invokevirtual` indicates that the actual implementation of the `setSalary()` method will be decided at runtime.

When the Java runtime tries to execute the above byte code instruction for the `emp.setSalary(salary)` statement, it finds that `emp` is referring to an object of the `Manager` class. The runtime looks for a `setSalary(D)V` method in the `Manager` class. Since the `Manager` class has a `setSalary(I)V` method (for `int`), the runtime does not find a matching method in the `Manager` class. Now, the runtime moves one level up in the class hierarchy and looks for a `setSalary(D)V` method in the `Employee` class and it finds it. So, the runtime decides to execute the implementation of the `setSalary(double)` method in the `Employee` class, and hence, you get the message “Inside Employee.setSalary()” printed on the standard output.

The declaration of the `setSalary(int)` method in the `Manager` class is an example of method overloading, and not method overriding. The `Manager` class has two `setSalary()` methods - a `setSalary(double)` inherited from the `Employee` class and a `setSalary(int)` declared in it. If you apply the logic discussed above, you should not have any problems in deciding which method will be called as part of a method call.

To give a new twist to our discussion, let us consider a little variation of the `Test` class, which we call `Test2` as listed in Listing 4.

Listing 4: Test2 class

```java
// Test2.java
package com.jdojo.blogs.methodbinding;

public class Test2 {
    public static void main(String[] args) {
        /* Create a new Manager object and assign its reference to a variable
         of Employee type */
        Employee emp = new Manager();
        
        /* Call the setSalary() method */
        int salary = 12000;
        ((Manager)emp).setSalary(salary); /* Which setSalary() method is called. Employee.setSalary() or Manager.setSalary() */
    }
}
```

Output:
Can you explain the output when the Test2 class is run? It is easy. The Test2 class has only one statement that is different from the Test class. It calls the setSalary(salary) method on the (Manager)emp expression instead of the emp expression. All you need to decide is - what is the compile-time type of the expression (Manager)emp. When you use a cast, the compile-time type of the expression is the cast type itself. That is, the compile-time type of the expression (Manager)emp is Manager. Therefore, the compiler will bind the setSalary(salary) method call to the setSalary(int) method of the Manager class. You can confirm this by dissembling the Test2 class using the javap tool, which outputs the following instruction. The last part setSalary(I)V means - a setSalary() method, which accepts an int (I for int) parameter and returns void (V for void).

17: invokevirtual #4 // Method com/jdojo/blogs/methodbinding/Manager.setSalary:(I)V

When the java runtime attempts to bind the setSalary(salary) call, it looks for a setSalary(I)V - starting from the Manager class and searching up the class hierarchy if not found. In this case, the runtime finds a setSalary(I)V method in the Manager class, and executes that method, which prints “Inside Manager.setSalary()”.

In the case of a static method, everything for a method call - the method signature and the method implementation, is decided by the compiler based on the compile-time type of the expression on which the static method call is made. With this knowledge in your armory, you can master the method binding mechanism for static methods yourself. You are advised to add a static method to the Employee class and hide it in the Manager class by declaring a static method with the same name. Call that static method in a class, say Test3. Use the javap tool to dissemble the code for Test3 class and see if the result matches your understanding of the method binding.
**Useful Links**

<table>
<thead>
<tr>
<th><strong>Author's website</strong></th>
<th><strong><a href="http://www.jdojo.com">www.jdojo.com</a></strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Contacting the Author</strong></td>
<td><strong><a href="mailto:ksharan@jdojo.com">ksharan@jdojo.com</a></strong></td>
</tr>
<tr>
<td><strong>Author's All Blogs</strong></td>
<td>**<a href="http://jdojo.com/feed/**">http://jdojo.com/feed/**</a></td>
</tr>
<tr>
<td><strong>Subscribing to Author's Blogs (RSS Feed)</strong></td>
<td><strong><a href="http://jdojo.com/feed/">http://jdojo.com/feed/</a></strong></td>
</tr>
<tr>
<td><strong>Downloading Sample Chapters of Harnessing Java 7 Book</strong></td>
<td><strong><a href="http://jdojo.com/sample-chapters/">http://jdojo.com/sample-chapters/</a></strong></td>
</tr>
<tr>
<td><strong>Purchasing Author's Books at Amazon</strong></td>
<td><strong><a href="http://www.amazon.com/Kishori-Sharan/e/B006Z81X7K/ref=ntt_atr_dp_pel_1">http://www.amazon.com/Kishori-Sharan/e/B006Z81X7K/ref=ntt_atr_dp_pel_1</a></strong></td>
</tr>
</tbody>
</table>