Core Java® SE 9

for the Impatient

Second Edition

Cay S. Horstmann

Core Java® SE 9 for the Impatient

Second Edition

For example, suppose the superclass `Employee` declares the instance variable `salary` as `protected` instead of `private`.

```
package com.horstmann.employees;

public class Employee {
    protected double salary;
    ...
}
```

All classes in the same package as `Employee` can access this field. Now consider a subclass from a different package:

```
package com.horstmann.managers;

import com.horstmann.employees.Employee;

public class Manager extends Employee {
    ...
    public double getSalary() {
        return salary + bonus; // OK to access protected salary variable
    }
}
```

The `Manager` class methods can peek inside the `salary` variable of `Manager` objects only, not of other `Employee` objects. This restriction is made so that you can't abuse the protected mechanism by forming subclasses just to gain access to protected features.

Of course, protected fields should be used with caution. Once provided, you cannot take them away without breaking classes that are using them.

Protected methods and constructors are more common. For example, the `clone` method of the `Object` class is protected since it is somewhat tricky to use (see Section 4.2.4, "Cloning Objects," page 151).

CAUTION: In Java, `protected` grants package-level access, and it only protects access from other packages.

4.1.10 Anonymous Subclasses

Just as you can have an anonymous class that implements an interface, you can have an anonymous class that extends a superclass. This can be handy for debugging:

```
ArrayList<String> names = new ArrayList<String>(100) {
    public void add(int index, String element) {
        super.add(index, element);
        System.out.printf("Adding %s at %d\n", element, index);
    }
};
```

The arguments in the parentheses following the superclass name are passed to the superclass constructor. Here, we construct an anonymous subclass of `ArrayList<String>` that overrides the `add` method. The instance is constructed with an initial capacity of 100.

A trick called *double brace initialization* uses the inner class syntax in a rather bizarre way. Suppose you want to construct an array list and pass it to a method:

```
ArrayList<String> friends = new ArrayList<>();
friends.add("Harry");
friends.add("Sally");
invite(friends);
```

If you won't ever need the array list again, it would be nice to make it anonymous. But then, how can you add the elements? Here is how:

```
invite(new ArrayList<String>() {{ add("Harry"); add("Sally"); }});
```

Note the double braces. The outer braces make an anonymous subclass of `ArrayList<String>`. The inner braces are an initialization block (see Chapter 2).

I am not recommending that you use this trick outside of Java trivia contests. There are several drawbacks beyond the confusing syntax. It is inefficient, and the constructed object can behave strangely in equality tests, depending on how the `equals` method is implemented.

4.1.11 Inheritance and Default Methods

Suppose a class extends a class and implements an interface, both of which happen to have a method of the same name.

```
public interface Named {
    default String getName() { return ""; }
}

public class Person {
    ...
    public String getName() { return name; }
}

public class Student extends Person implements Named {
    ...
}
```

In this situation, the superclass implementation always wins over the interface implementation. There is no need for the subclass to resolve the conflict.

In contrast, as you saw in Chapter 3, you must resolve a conflict when the same default method is inherited from two interfaces.

The "classes win" rule ensures compatibility with Java 7. If you add default methods to an interface, it has no effect on code that worked before there were default methods.

4.1.12 Method Expressions with `super`

Recall from Chapter 3 that a method expression can have the form *object*::*instanceMethod*. It is also valid to use `super` instead of an object reference. The method expression

```
super::instanceMethod
```

uses `this` as the target and invokes the superclass version of the given method. Here is an artificial example that shows the mechanics:

```
public class Worker {
    public void work() {
        for (int i = 0; i < 100; i++) System.out.println("Working");
    }
}

public class ConcurrentWorker extends Worker {
    public void work() {
        Thread t = new Thread(super::work);
        t.start();
    }
}
```

The thread is constructed with a `Runnable` whose `run` method calls the `work` method of the superclass.

4.2 `Object`: The Cosmic Superclass

Every class in Java directly or indirectly extends the class `Object`. When a class has no explicit superclass, it implicitly extends `Object`. For example,

```
public class Employee { ... }
```

is equivalent to

```
public class Employee extends Object { ... }
```

The `Object` class defines methods that are applicable to any Java object (see Table 4-1). We will examine several of these methods in detail in the following sections.

NOTE: Arrays are classes. Therefore, it is legal to convert an array, even a primitive type array, to a reference of type `Object`.

Table 4–1 The Methods of the `java.lang.Object` Class

Method	Description
`String toString()`	Yields a string representation of this object, by default the name of the class and the hash code.
`boolean equals(Object other)`	Returns `true` if this object should be considered equal to `other`, `false` if `other` is `null` or different from `other`. By default, two objects are equal if they are identical. Instead of `obj.equals(other)`, consider the null-safe alternative `Objects.equals(obj, other)`.
`int hashCode()`	Yields a hash code for this object. Equal objects must have the same hash code. Unless overridden, the hash code is assigned in some way by the virtual machine.
`Class<?> getClass()`	Yields the `Class` object describing the class to which this object belongs.
`protected Object clone()`	Makes a copy of this object. By default, the copy is shallow.
`protected void finalize()`	This method is called when this object is reclaimed by the garbage collector. Don't override it.
`wait, notify, notifyAll`	See Chapter 10.

4.2.1 The `toString` Method

An important method in the `Object` class is the `toString` method that returns a string description of an object. For example, the `toString` method of the `Point` class returns a string like this:

```
java.awt.Point[x=10,y=20]
```

Many `toString` methods follow this format: the name of the class, followed by the instance variables enclosed in square brackets. Here is such an implementation of the `toString` method of the `Employee` class:

```
public String toString() {
    return getClass().getName() + "[name=" + name
        + ",salary=" + salary + "]";
}
```

By calling `getClass().getName()` instead of hardwiring the string `"Employee"`, this method does the right thing for subclasses as well.

In a subclass, call `super.toString()` and add the instance variables of the subclass, in a separate pair of brackets:

```
public class Manager extends Employee {
    ...
    public String toString() {
        return super.toString() + "[bonus=" + bonus + "]";
    }
}
```

Whenever an object is concatenated with a string, the Java compiler automatically invokes the `toString` method on the object. For example:

```
Point p = new Point(10, 20);
String message = "The current position is " + p;
    // Concatenates with p.toString()
```

TIP: Instead of writing `x.toString()`, you can write `"" + x`. This expression even works if `x` is `null` or a primitive type value.

The `Object` class defines the `toString` method to print the class name and the hash code (see Section 4.2.3, "The `hashCode` Method," page 150). For example, the call

```
System.out.println(System.out)
```

produces an output that looks like `java.io.PrintStream@2f6684` since the implementor of the `PrintStream` class didn't bother to override the `toString` method.

CAUTION: Arrays inherit the `toString` method from `Object`, with the added twist that the array type is printed in an archaic format. For example, if you have the array

```
int[] primes = { 2, 3, 5, 7, 11, 13 };
```

then `primes.toString()` yields a string such as `"[I@1a46e30"`. The prefix `[I` denotes an array of integers.

The remedy is to call `Arrays.toString(primes)` instead, which yields the string `"[2, 3, 5, 7, 11, 13]"`. To correctly print multidimensional arrays (that is, arrays of arrays), use `Arrays.deepToString`.

4.2.2 The equals Method

The equals method tests whether one object is considered equal to another. The equals method, as implemented in the Object class, determines whether two object references are identical. This is a pretty reasonable default—if two objects are identical, they should certainly be equal. For quite a few classes, nothing else is required. For example, it makes little sense to compare two Scanner objects for equality.

Override the equals method only for state-based equality testing, in which two objects are considered equal when they have the same contents. For example, the String class overrides equals to check whether two strings consist of the same characters.

CAUTION: Whenever you override the equals method, you *must* provide a compatible hashCode method as well—see Section 4.2.3, "The hashCode Method" (page 150).

Suppose we want to consider two objects of a class Item equal if their descriptions and prices match. Here is how you can implement the equals method:

```
public class Item {
    private String description;
    private double price;
    ...
    public boolean equals(Object otherObject) {
        // A quick test to see if the objects are identical
        if (this == otherObject) return true;

        // Must return false if the parameter is null
        if (otherObject == null) return false;
        // Check that otherObject is an Item
        if (getClass() != otherObject.getClass()) return false;
        // Test whether the instance variables have identical values
        Item other = (Item) otherObject;
        return Objects.equals(description, other.description)
            && price == other.price;
    }

    public int hashCode() { ... } // See Section 4.2.3
}
```

There are a number of routine steps that you need to go through in an equals method:

1. It is common for equal objects to be identical, and that test is very inexpensive.