Supplement: Methods in the Object Class
For Introduction to Java Programming
By Y. Daniel Liang

Chapter 11 introduced the **toString** method in the **Object** class. Other useful methods are as follows:

- **public boolean equals(Object object)**
- **public int hashCode()**
- **protected void finalize() throws Throwable**
- **protected native Object clone()**
  throws CloneNotSupportedException
- **public final native Class getClass()**

NOTE
The **native** modifier indicates that the method is implemented using a programming language other than Java. Some methods, such as **clone**, need to access hardware using the native machine language or the C language. These methods are marked **native**. A native method can be **final**, **public**, **private**, **protected**, overloaded, or overridden.

NOTE
The **finalize** method may throw **Throwable**, and the **clone** method may throw **CloneNotSupportedException**. Exception handling will be introduced in Chapter 18, “Exceptions and Assertions.” (*Chapter 18 can be covered after this chapter.*) For now, you need to know that **throws Throwable** and **throws CloneNotSupportedException** are part of the method declarations for the **finalize** and **clone** methods.

*1 The **equals** Method*

The **equals** method tests whether two objects are equal. The syntax for invoking it is:

```
  object1.equals(object2);
```

The default implementation of the **equals** method in the **Object** class is:

```
  public boolean equals(Object obj) {
    return (this == obj);
  }
```

Thus, using the **equals** method is equivalent to the **==** operator in the **Object** class, but it is really intended for the subclasses of the **Object** class to modify the **equals** method to test whether two distinct objects have the same content.

You have already used the **equals** method to compare two strings in §8.2, "The **String** Class." The **equals** method in the **String** class is inherited from the **Object** class and is modified in the **String** class to test whether two strings are identical in content. You can override the **equals** method in the **Circle** class to compare whether two circles are equal based on their radius as follows:

```
public boolean equals(Object o) {
  if (o instanceof Circle) {
    return radius == ((Circle)o).radius;
  }
  else
    return false;
}
```

NOTE
The **==** comparison operator is used for comparing two primitive data type values or for determining whether two objects have the same references. The **equals** method is intended to test whether two objects have the same contents, provided that the method is modified in the defining class of the objects. The **==** operator is stronger than the **equals** method, in that the **==** operator checks whether the two reference variables refer to the same object.

CAUTION
Using the signature **equals(SomeClassName obj)** (e.g., **equals(Circle c)**) to override the **equals** method in a subclass is a common mistake. You should use **equals(Object obj)**). See Review Question 10.16.

*2 The **hashCode** Method*

Invoking **hashCode()** on an object returns the object's hash code. *Hash code* is an integer that can be used to store the object in a hash set so that it can be located quickly. Hash sets will be introduced in Chapter 26, "Java Collections Framework." The **hashCode** implemented in the **Object** class returns the internal memory address of the object in hexadecimal. Your class should override the **hashCode** method whenever the **equals** method is overridden. By contract, if two objects are equal, their hash codes must be same. Two unequal objects may have the same hash code, but you should implement the **hashCode** method to avoid too many such cases. Additionally, invoking the **hashCode** method multiple times must return the same integer during one execution of the program. The integer need not be the same in different executions. For example, the **hashCode** method is overridden in the **String** class by returning $s_0*31^{(n-1)} + s_1*31^{(n-2)} + ... + s_{n-1}$ as the hash code, where s_i is **s.charAt(i)**.

If you override the **equals** method in a class, you should also override the hashCode method in the same class to ensure that two equal **Circle** objects have the same hashCode. For example, you may override the **hashCode** method in the **Circle** class as follows:

```
public int hashCode() {
  return (int)(radius * 1999711);
}
```

3 The *finalize* Method

The **finalize** method is invoked on an object by the garbage collector when the object becomes garbage. An object becomes garbage if it is no longer accessed. By default, the **finalize** method does nothing. A subclass should override the **finalize** method to dispose of system resources or to perform other cleanup, if necessary.

> NOTE: The **finalize** method is invoked by the JVM. You should never write the code to invoke it in your program. For this reason, the protected modifier is appropriate.

Listing 10.9 demonstrates the effect of overriding the **finalize()** method.

Listing 1 FinalizationDemo.java

```
public class FinalizationDemo {
  public static void main(String[] args) {
    Cake a1 = new Cake(1);
    Cake a2 = new Cake(2);
    Cake a3 = new Cake(3);

    // To dispose the objects a2 and a3
    a2 = a3 = null;
    System.gc(); // Invoke the Java garbage collector
  }
}

class Cake extends Object {
  private int id;

  public Cake(int id) {
    this.id = id;
    System.out.println("Cake object " + id + " is created");
  }

  protected void finalize() throws java.lang.Throwable {
    super.finalize();
    System.out.println("Cake object " + id + " is disposed");
  }
}
```

Sample Output

```
     Cake object 1 is created
     Cake object 2 is created
     Cake object 3 is created
     Cake object 2 is disposed
     Cake object 3 is disposed
```

Line 8 assigns **null** to **a2** and **a3**. The objects previously referenced by **a2** and **a3** are no longer accessible. Therefore, they are garbage. **System.gc()** in line 9 requests the garbage collector to be invoked to reclaim space from all discarded objects. Normally you don't need to invoke this method explicitly, because the JVM automatically invokes it whenever necessary. The **finalize** method on the objects **a2** and **a3** are invoked by the garbage collector. When the program terminates, **a1** also becomes garbage, and **a1**'s **finalize** method is then invoked. Since the program has already exited, no message is displayed on the console.

Line 22 invokes the **finalize()** method in the superclass. This is a good practice to ensure that the finalization operations defined in the superclass are carried out.

*4 The **clone** Method*

Sometimes you need to make a copy of an object. Mistakenly, you might use the assignment statement, as follows:

```
  newObject = someObject;
```

This statement does not create a duplicate object. It simply assigns the reference of **someObject** to **newObject**. To create a new object with separate memory space, use the **clone()** method:

```
  newObject = someObject.clone();
```

This statement copies **someObject** to a new memory location and assigns the reference of the new object to **newObject**. For example,

```
  java.util.Date date = new java.util.Date();
  java.util.Date date1 = (java.util.Date)(date.clone());
```

creates a new **Date** object, **date**, and its clone **date1**. Note that **date.equals(date1)** is true, but **date == date1** is false.

NOTE
Not all objects can be cloned. For an object to be cloneable, its class must implement the **java.lang.Cloneable** interface, which is introduced in §10.4.4, "The **Cloneable** Interface."

TIP
An array is treated as an object in Java and is an instance of the **Object** class. The **clone** method can also be used to copy arrays. The following statement uses the **clone** method to copy the **sourceArray** of the **int[]** type to the **targetArray**:

```
  int[] targetArray = (int[])sourceArray.clone();
```

Since the return type of the **clone** method is **Object**, **(int[])** is used to cast it to the **int[]** type.

*5 The **getClass** Method*

A class must be loaded in order to be used. When the JVM loads the class, it creates an object that contains the information about the class, such as class name, constructors, and methods. This object is an instance of **java.lang.Class**. It is referred to as a *meta-object* in this book, because it describes the information about the class.

Through the meta-object, you can discover the information about the class at runtime. Every object can use the **getClass()** method to return its meta-object. For example, the following code

```
Object obj = new Object();
Class metaObject = obj.getClass();

System.out.println("Object obj's class is "
  + metaObject.getName());
```

displays

```
  Object obj's class is java.lang.Object
```

NOTE: There is only one meta-object for a class. Every object has a meta-object. If two objects were created from the same class, their meta-objects are the same.