Chapter 14

The Object Class

Chapter 13 examined a few of the hundreds of classes available in Java’s standard library. One standard Java class that was not mentioned deserves special attention. It rarely is used directly by programmers, but all of Java’s classes, whether standard or programmer-defined, depend on it for some of their basic functionality. This chapter is devoted to the Object class, the root of Java’s class hierarchy.

14.1 Class Object

While a class may serve as the superclass for any number of subclasses, every class, except for one, has exactly one superclass. The standard Java class Object has no superclass. Object is at the root of the Java class hierarchy. When a class is defined without the extends reserved word, the class implicitly extends class Object. Thus, the following definition

```java
public class Rational {
    /* Details omitted . . . */
}
```

is actually just a shorthand for

```java
public class Rational extends Object {
    /* Details omitted . . . */
}
```

Since Object is such a fundamental class it is found in the java.lang package. Its fully qualified name, therefore, is java.lang.Object.

The is a relationship is transitive like many mathematical relations. In mathematics, for any real numbers x, y, and z, if \(x < y\) and \(y < z\), we know \(x < z\). This is known as the transitive property of the less than relation. In Java, if class \(Z\) extends class \(Y\), and class \(Y\) extends class \(X\), we know \(\text{a } Z\text{ is a } Y\) and \(\text{a } Y\text{ is a } X\). The is a relation is transitive, so we also know that \(\text{a } Z\text{ is a } X\). Since Object is at the top of Java’s class hierarchy, because of the transitive nature of inheritance, any object, no matter what its type, is a Object. Therefore, any reference type can be passed to method that specifies Object as a formal parameter or be assigned to any variable with a declared type of Object.
14.2 Object Methods

Since every class except `Object` has `Object` as a superclass (either directly or indirectly), every class inherits the methods provided by the `Object` class. The `Object` class provides 11 methods, and two of the 11 `Object` methods are highlighted in Table 14.1.

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>equals</code></td>
<td>Determines if two objects are to be considered “equal”</td>
</tr>
<tr>
<td><code>toString</code></td>
<td>Returns the string representation for an object</td>
</tr>
</tbody>
</table>

Table 14.1: Two commonly used `Object` methods

The `equals()` method inherited from `Object` checks for reference equality; that is, two references are equal if and only if they refer to exactly the same object. This performs the same test as the `==` operator when it is applied to reference types. In general, `==` is not appropriate for comparing object references in most applications. Consider `EqualsTest` (14.1) which illustrates the differences between `equals()` and `==`,

```java
class EqualsTest {
    public static void main(String[] args) {
        String s1 = "hello",
        s2 = new String(s1), // s2 is new string, a copy of s1
        s3 = s1; // s3 is an alias of s1
        System.out.println("s1 = " + s2 + ", s2 = " + s2 + ", s3 = " + s3);
        if ( s1 == s2 ) {
            System.out.println("s1 == s2");
        } else {
            System.out.println("s1 != s2");
        }
        if ( s1.equals(s2) ) {
            System.out.println("s1 equals s2");
        } else {
            System.out.println("s1 does not equal s2");
        }
        if ( s1 == s3 ) {
            System.out.println("s1 == s3");
        } else {
            System.out.println("s1 != s3");
        }
        if ( s1.equals(s3) ) {
            System.out.println("s1 equals s3");
        } else {
            System.out.println("s1 does not equal s3");
        }
        if ( s2 == s3 ) {
            System.out.println("s2 == s3");
        } else {
            System.out.println("s2 != s3");
        }
        if ( s2.equals(s3) ) {
```
The output of `EqualsTest` is

\[
\begin{align*}
\text{s1} & = \text{hello}, \text{s2} = \text{hello}, \text{s3} = \text{hello} \\
\text{s1} \neq \text{s2} & \\
\text{s1} & \text{ equals s2} \\
\text{s1} \equiv \text{s3} & \\
\text{s1} & \text{ equals s3} \\
\text{s2} \neq \text{s3} & \\
\text{s2} & \text{ equals s3}
\end{align*}
\]

The `equals()` method correctly determined that all three strings contain the same characters; the `==` operator correctly shows that `s1` and `s3` refer to exactly the same `String` object.

The `toString()` method is called when an object must be represented as a string. Passing an object reference to a `System.out.println()` call causes the compiler to generate code that calls the object’s `toString()` method so “the object” can be printed on the screen. The `toString()` method inherited from `Object` does little more than return a string that includes the name of the object’s class and a numeric code that is related to that object’s address in memory. The numeric code is expressed in hexadecimal notation, base 16. Hexadecimal numbers are composed of the digits

\[
0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f
\]

(See § 4.1.) This numeric code is of little use to applications programmers except to verify that two references point to distinct objects. If the hexadecimal codes for two references differ, then the objects to which they refer must be different. Subclasses should override the `toString()` method to provide clients more useful information.

These two methods would be valuable additions to the `RationalNumber` class (§ 7.5). The overridden `equals()` method would check to see if the numerator and denominator of another fraction, in reduced form, is equal to a given fraction, in reduced form. The overridden `toString()` method returns a string that humans can easily interpret as a fraction (for example, \( \frac{1}{2} \) becomes "1/2". These methods could be written:

```java
public boolean equals(Object r) {
    if ( r != null && r instanceof Rational ) {
        Rational thisRational = reduce(),
            thatRational = ((Rational) r).reduce();
        return thisRational.numerator == thatRational.numerator &&
                thisRational.denominator == thatRational.denominator;
    }
    return false;
}
```
The `toString()` method is straightforward, but the `equals()` method requires some comment. The `equals()` method as defined in `Object` will accept a reference to any object, since any Java object is a `Object`. To properly override the method, the parameter must be declared to be of type `Object`. This means our method needs to ensure the actual parameter is of type `Rational`, and, if it is, it must be cast to type `Rational`. Another issue is that `null` is a literal value that can be assigned to any object reference, we must provide for the possibility that the client may try to pass a `null` reference. The client may do this either directly:

```java
Rational frac = new Rational(1, 2);
if (frac.equals(null)) { /* Do whatever . . . */}
```

or indirectly:

```java
Rational frac1 = new Rational(1, 2), frac2 = makeFract();
// The method makeFract() might return null, but the client
// may be oblivious to the fact.
if (frac1.equals(frac2)) { /* Do whatever . . . */}
```

Since the parameter’s type is `Object`, the following code is legal:

```java
Rational frac = new Rational(1, 2);
TrafficLightModel light = new TrafficLightModel(TrafficLightModel.RED);
if (frac.equals(light)) { /* Do whatever . . . */}
```

We probably never want a `Rational` object to be considered equal to a `TrafficLightModel` object, so we need to be sure our method would handle this situation gracefully. The `instanceof` operator returns `true` if a given object reference refers to an instance of the given class. The `instanceof` operator respects the `is a` relationship as `InstanceOfTest` (14.2) demonstrates.

```java
class X {}
class Y extends X {}
class Z {}

public class InstanceOfTest {
    public static void main(String[] args) {
        Y y = new Y();  // Make a new Y object
        Object z = new Z();  // Make a new Z object
        if (y instanceof Y) {
            System.out.println("y is an instance of Y");
        } else {
            System.out.println("y is NOT an instance of Y");
        }
        if (y instanceof X) {
```
System.out.println("y is an instance of X");
} else {
    System.out.println("y is NOT an instance of X");
}
if ( z instanceof Y ) {
    System.out.println("z is an instance of Y");
} else {
    System.out.println("z is NOT an instance of Y");
}
}

Listing 14.2: InstanceOfTest—demonstrates the use of the instanceof operator

The output of InstanceOfTest is

```
y is an instance of Y
y is an instance of X
z is NOT an instance of Y
x is NOT an instance of Y
```

In InstanceOfTest, y is an instance of Y and also an instance of X; said another way, *y is a Y and y is a X*. On the other hand, z is not an instance of Y. Note that z was declared as an Object reference, not a Z reference. If z’s declared type is Z, then the compiler will generate an error for the expression z instanceof Y since it can deduce that z can never be an instance of Y. The Object type is assignment compatible with all other types so the expression z instanceof Y is not illegal when z’s declared type is a supertype of Y. Since the declared type of z and x are, respectively, Object and X, and both Object and X are supertypes of Y, all of the uses of the instanceof operator in InstanceOfTest are legal.

14.3 Summary

- **Object** in the class java.lang is the superclass of all Java classes, even programmer-defined classes.
- The *is a* relation is transitive, so any object reference *is a Object*.
- The instanceof reserved word is an operator used to determine if an object reference *is a* given type. The instanceof operator respects the transitivity of the *is a* relation.
- Of the 11 methods provided by Object, equals() and toString() are often overridden by subclasses.
- An object’s toString() method is called when the compiler must produce a string representation of the object, such as for a call to System.out.println().
- The equals() method’s formal parameter in Object. Subclasses that override equals() must ensure that the actual parameter is non-null and has the exact type of the subclass itself.
- The programmer has complete control over what the overridden toString() and equals() methods do, but their behavior should agree with the natural expectations of the client.
14.4 Exercises

1. Modify the traffic light view classes, TextTrafficLight (10.2) and TextTurnLight (11.2) so that toString() replaces the show() methods. This will allow code originally written as

```
System.out.println(light.show());
```

to be written more simply as

```
System.out.println(light);
```

with the same behavior. Can you think of a way to make it work by modifying only the superclass, TextTrafficLight?