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

Click here to view code image

```
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.

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.

Note

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

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:

Click here to view code image

```
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:

Click here to view code image

```
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:

Click here to view code image

```
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

Click here to view code image

```
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

Click here to view code image

```
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:

Click here to view code image

```
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.
2. Every `equals` method is required to return `false` when comparing against

null.

3. Since the equals method overrides Object.equals, its parameter is of type Object, and you need to cast it to the actual type so you can look at its instance variables. Before doing that, make a type check, either with the getClass method or with the instanceof operator.
4. Finally, compare the instance variables. Use == for primitive types. However, for double values, if you are concerned about ±∞ or NaN, use Double.equals. For objects, use Objects.equals, a null-safe version of the equals method. The call Objects.equals(x, y) returns false if x is null, whereas x.equals(y) would throw an exception.

Tip

If you have instance variables that are arrays, use the static Arrays.equals method to check that the arrays have equal length and the corresponding array elements are equal.

When you define the equals method for a subclass, first call equals on the superclass. If that test doesn't pass, the objects can't be equal. If the instance variables of the superclass are equal, then you are ready to compare the instance variables of the subclass.

Click here to view code image

```
public class DiscountedItem extends Item {
private double discount;
...
public boolean equals(Object otherObject) {
if (!super.equals(otherObject)) return false;
DiscountedItem other = (DiscountedItem) otherObject;
return discount == other.discount;
}

public int hashCode() { ... }
}
```

Note that the getClass test in the superclass fails if otherObject is not a DiscountedItem.

How should the equals method behave when comparing values that belong to different classes? This has been an area of some controversy. In the preceding example, the equals method returns false if the classes don't match exactly. But many programmers use an instanceof test instead:

Click here to view code image

```
if (!(otherObject instanceof Item)) return false;
```

This leaves open the possibility that `otherObject` can belong to a subclass. For example, you can compare an `Item` with a `DiscountedItem`.

However, that kind of comparison doesn't usually work. One of the requirements of the `equals` method is that it is *symmetric*: For non-null `x` and `y`, the calls `x.equals(y)` and `y.equals(x)` need to return the same value.

Now suppose `x` is an `Item` and `y` a `DiscountedItem`. Since `x.equals(y)` doesn't consider discounts, neither can `y.equals(x)`.

Note

The Java API contains over 150 implementations of `equals` methods, with a mixture of `instanceof` tests, calling `getClass`, catching a `ClassCastException`, or doing nothing at all. Check out the documentation of the `java.sql.Timestamp` class, where the implementors note with some embarrassment that the `Timestamp` class inherits from `java.util.Date`, whose `equals` method uses an `instanceof` test, and it is therefore impossible to override `equals` to be both symmetric and accurate.

There is one situation where the `instanceof` test makes sense: if the notion of equality is fixed in the superclass and never varies in a subclass. For example, this is the case if we compare employees by ID. In that case, make an `instanceof` test and declare the `equals` method as `final`.

Click here to view code image

```
public class Employee {
private int id;
...
public final boolean equals(Object otherObject) {
if (this == otherObject) return true;
if (!(otherObject instanceof Employee)) return false;
Employee other = (Employee) otherObject;
return id == other.id;
}

public int hashCode() { ... }
}
```

4.2.3 The `hashCode` Method

A *hash code* is an integer that is derived from an object. Hash codes should be scrambled—if `x` and `y` are two unequal objects, there should be a high probability that `x.hashCode()` and `y.hashCode()` are different. For example, `"Mary".hashCode()` is 2390779, and `"Myra".hashCode()` is 2413819.

The `String` class uses the following algorithm to compute the hash code:

Click here to view code image

```
int hash = 0;
for (int i = 0; i < length(); i++)
hash = 31 * hash + charAt(i);
```

The `hashCode` and `equals` methods must be *compatible*: If `x.equals(y)`, then it must be the case that `x.hashCode() == y.hashCode()`. As you can see, this is the case for the `String` class since strings with equal characters produce the same hash code.

The `Object.hashCode` method derives the hash code in some implementation-dependent way. It can be derived from the object's memory location, or a number (sequential or pseudorandom) that is cached with the object, or a combination of both. Since `Object.equals` tests for identical objects, the only thing that matters is that identical objects have the same hash code.

If you redefine the `equals` method, you will also need to redefine the `hashCode` method to be compatible with `equals`. If you don't, and users of your class insert objects into a hash set or hash map, they might get lost!

In your `hashCode` method, simply combine the hash codes of the instance variables. For example, here is a `hashCode` method for the `Item` class:

Click here to view code image

```
class Item {
...
public int hashCode() {
return Objects.hash(description, price);
}
}
```

The `Objects.hash` varargs method computes the hash codes of its arguments and combines them. The method is null-safe.

If your class has instance variables that are arrays, compute their hash codes first with the static `Arrays.hashCode` method, which computes a hash code composed of the hash codes of the array elements. Pass the result to `Objects.hash`.

Caution

In an interface, you can never make a default method that redefines one of the methods in the `Object` class. In particular, an interface can't define a default method for `toString`, `equals`, or `hashCode`. As a consequence of the "classes win" rule (see Section 4.1.11, "Inheritance and Default Methods," page 144), such a method could never win against `Object.toString`, `Object.equals`, or `Object.hashCode`.

4.2.4 Cloning Objects

You have just seen the "big three" methods of the `Object` class that are commonly overridden: `toString`, `equals`, and `hashCode`. In this section, you will learn how to override the `clone` method. As you will see, this is complex, and it is also rarely necessary. Don't override `clone` unless you have a good reason to do so. Less than five percent of the classes in the standard Java library implement `clone`.

The purpose of the `clone` method is to make a "clone" of an object—a distinct object with the same state of the original. If you mutate one of the objects, the other stays unchanged.

Click here to view code image

```
Employee cloneOfFred = fred.clone();
cloneOfFred.raiseSalary(10); // fred unchanged
```

The `clone` method is declared as `protected` in the `Object` class, so you must override it if you want users of your class to clone instances.

The `Object.clone` method makes a *shallow copy*. It simply copies all instance variables from the original to the cloned object. That is fine if the variables are primitive or immutable. But if they aren't, then the original and the clone share mutable state, which can be a problem.

Consider a class for email messages that has a list of recipients.

Click here to view code image

```
public final class Message {
private String sender;
private ArrayList<String> recipients;
private String text;
...
public void addRecipient(String recipient) { ... };
}
```

If you make a shallow copy of a `Message` object, both the original and the clone share the `recipients` list (see Figure 4-1):

Click here to view code image

```
Message specialOffer = ...;
Message cloneOfSpecialOffer = specialOffer.clone();
```

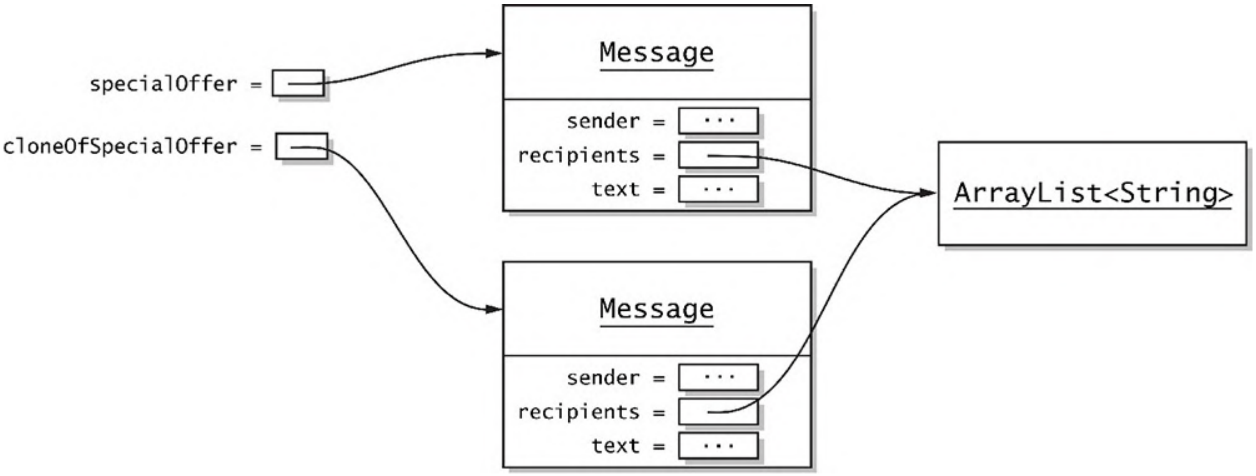

Figure 4-1 A shallow copy of an object

If either object changes the recipient list, the change is reflected in the other. Therefore, the `Message` class needs to override the `clone` method to make a *deep copy*.

It may also be that cloning is impossible or not worth the trouble. For example, it would be very challenging to clone a `Scanner` object.

In general, when you implement a class, you need to decide whether

1. You do not want to provide a `clone` method, or
2. The inherited `clone` method is acceptable, or
3. The `clone` method should make a deep copy.

For the first option, simply do nothing. Your class will inherit the `clone` method, but no user of your class will be able to call it since it is `protected`.

To choose the second option, your class must implement the `Cloneable` interface. This is an interface without any methods, called a *tagging* or *marker* interface. (Recall that the `clone` method is defined in the `Object` class.) The `Object.clone` method checks that this interface is implemented before making a shallow copy, and throws a `CloneNotSupportedException` otherwise.

You will also want to raise the scope of `clone` from `protected` to `public`, and change the return type.

Finally, you need to deal with the `CloneNotSupportedException`. This is a *checked* exception, and as you will see in Chapter 5, you must either declare or catch it. If your class is `final`, you can catch it. Otherwise, declare the exception since it is possible that a subclass might again want to throw it.

Click here to view code image

```
public class Employee implements Cloneable {
...
public Employee clone() throws CloneNotSupportedException {
return (Employee) super.clone();
}
}
```

The cast `(Employee)` is necessary since the return type of `Object.clone` is `Object`.

The third option for implementing the `clone` method, in which a class needs to make a deep copy, is the most complex case. You don't need to use the `Object.clone` method at all. Here is a simple implementation of `Message.clone`:

Click here to view code image

```
public Message clone() {
Message cloned = new Message(sender, text);
cloned.recipients = new ArrayList<>(recipients);
return cloned;
}
```

Alternatively, you can call `clone` on the superclass and the mutable instance variables.

The `ArrayList` class implements the `clone` method, yielding a shallow copy. That is, the original and cloned list share the element references. That is fine in our case since the elements are strings. If not, we would have had to clone each element as well.

However, for historical reasons, the `ArrayList.clone` method has return type `Object`. You need to use a cast.

Click here to view code image

```
cloned.recipients = (ArrayList<String>) recipients.clone(); // Warning
```

Unhappily, as you will see in Chapter 6, that cast cannot be fully checked at runtime, and you will get a warning. You can suppress the warning with an annotation, but that annotation can only be attached to a declaration (see Chapter 12). Here is the complete method implementation:

Click here to view code image

```
public Message clone() {
try {
Message cloned = (Message) super.clone();
@SuppressWarnings("unchecked") ArrayList<String> clonedRecipients
= (ArrayList<String>) recipients.clone();
cloned.recipients = clonedRecipients;
return cloned;
} catch (CloneNotSupportedException ex) {
return null; // Can't happen
}
}
```

In this case, the `CloneNotSupportedException` cannot happen since the `Message` class is `Cloneable` and `final`, and `ArrayList.clone` does not throw the exception.

Note

Arrays have a public `clone` method whose return type is the same as the type of the array. For example, if `recipients` had been an array, not an array list, you could have cloned it as

Click here to view code image

```
cloned.recipients = recipients.clone(); // No cast required
```

4.3 Enumerations

You saw in Chapter 1 how to define enumerated types. Here is a typical example, defining a type with exactly four instances:

Click here to view code image

```
public enum Size { SMALL, MEDIUM, LARGE, EXTRA_LARGE };
```

In the following sections, you will see how to work with enumerations.