CHAPTER 3

Methods Common to All Objects

ALTHOUGH `Object` is a concrete class, it is designed primarily for extension. All of its nonfinal methods (`equals`, `hashCode`, `toString`, `clone`, and `finalize`) have explicit *general contracts* because they are designed to be overridden. It is the responsibility of any class overriding these methods to obey their general contracts; failure to do so will prevent other classes that depend on the contracts (such as `HashMap` and `HashSet`) from functioning properly in conjunction with the class.

This chapter tells you when and how to override the nonfinal `Object` methods. The `finalize` method is omitted from this chapter because it was discussed in Item 8. While not an `Object` method, `Comparable.compareTo` is discussed in this chapter because it has a similar character.

Item 10: Obey the general contract when overriding `equals`

Overriding the `equals` method seems simple, but there are many ways to get it wrong, and consequences can be dire. The easiest way to avoid problems is not to override the `equals` method, in which case each instance of the class is equal only to itself. This is the right thing to do if any of the following conditions apply:

- **Each instance of the class is inherently unique.** This is true for classes such as `Thread` that represent active entities rather than values. The `equals` implementation provided by `Object` has exactly the right behavior for these classes.
- **There is no need for the class to provide a "logical equality" test.** For example, `java.util.regex.Pattern` could have overridden `equals` to check whether two `Pattern` instances represented exactly the same regular expression, but the designers didn't think that clients would need or want this functionality. Under these circumstances, the `equals` implementation inherited from `Object` is ideal.

- **A superclass has already overridden `equals`, and the superclass behavior is appropriate for this class.** For example, most `Set` implementations inherit their `equals` implementation from `AbstractSet`, `List` implementations from `AbstractList`, and `Map` implementations from `AbstractMap`.
- **The class is private or package-private, and you are certain that its `equals` method will never be invoked.** If you are extremely risk-averse, you can override the `equals` method to ensure that it isn't invoked accidentally:

```
@Override public boolean equals(Object o) {
    throw new AssertionError(); // Method is never called
}
```

So when is it appropriate to override `equals`? It is when a class has a notion of *logical equality* that differs from mere object identity and a superclass has not already overridden `equals`. This is generally the case for *value classes.* A value class is simply a class that represents a value, such as `Integer` or `String`. A programmer who compares references to value objects using the `equals` method expects to find out whether they are logically equivalent, not whether they refer to the same object. Not only is overriding the `equals` method necessary to satisfy programmer expectations, it enables instances to serve as map keys or set elements with predictable, desirable behavior.

One kind of value class that does *not* require the `equals` method to be overridden is a class that uses instance control (Item 1) to ensure that at most one object exists with each value. Enum types (Item 34) fall into this category. For these classes, logical equality is the same as object identity, so `Object`'s `equals` method functions as a logical `equals` method.

When you override the `equals` method, you must adhere to its general contract. Here is the contract, from the specification for `Object` :

The `equals` method implements an *equivalence relation.* It has these properties:

- *Reflexive*: For any non-null reference value `x`, `x.equals(x)` must return `true`.
- *Symmetric*: For any non-null reference values `x` and `y`, `x.equals(y)` must return `true` if and only if `y.equals(x)` returns `true`.
- *Transitive*: For any non-null reference values `x`, `y`, `z`, if `x.equals(y)` returns `true` and `y.equals(z)` returns `true`, then `x.equals(z)` must return `true`.
- *Consistent*: For any non-null reference values `x` and `y`, multiple invocations of `x.equals(y)` must consistently return `true` or consistently return `false`, provided no information used in `equals` comparisons is modified.
- For any non-null reference value `x`, `x.equals(null)` must return `false`.

Unless you are mathematically inclined, this might look a bit scary, but do not ignore it! If you violate it, you may well find that your program behaves erratically or crashes, and it can be very difficult to pin down the source of the failure. To paraphrase John Donne, no class is an island. Instances of one class are frequently passed to another. Many classes, including all collections classes, depend on the objects passed to them obeying the `equals` contract.

Now that you are aware of the dangers of violating the `equals` contract, let's go over the contract in detail. The good news is that, appearances notwithstanding, it really isn't very complicated. Once you understand it, it's not hard to adhere to it.

So what is an equivalence relation? Loosely speaking, it's an operator that partitions a set of elements into subsets whose elements are deemed equal to one another. These subsets are known as *equivalence classes*. For an `equals` method to be useful, all of the elements in each equivalence class must be interchangeable from the perspective of the user. Now let's examine the five requirements in turn:

Reflexivity—The first requirement says merely that an object must be equal to itself. It's hard to imagine violating this one unintentionally. If you were to violate it and then add an instance of your class to a collection, the `contains` method might well say that the collection didn't contain the instance that you just added.

Symmetry—The second requirement says that any two objects must agree on whether they are equal. Unlike the first requirement, it's not hard to imagine violating this one unintentionally. For example, consider the following class, which implements a case-insensitive string. The case of the string is preserved by `toString` but ignored in `equals` comparisons:

```
// Broken - violates symmetry!
public final class CaseInsensitiveString {
    private final String s;

    public CaseInsensitiveString(String s) {
        this.s = Objects.requireNonNull(s);
    }

    // Broken - violates symmetry!
    @Override public boolean equals(Object o) {
        if (o instanceof CaseInsensitiveString)
            return s.equalsIgnoreCase(
                ((CaseInsensitiveString) o).s);
        if (o instanceof String)  // One-way interoperability!
            return s.equalsIgnoreCase((String) o);
        return false;
    }
    ...  // Remainder omitted
}
```

The well-intentioned `equals` method in this class naively attempts to interoperate with ordinary strings. Let's suppose that we have one case-insensitive string and one ordinary one:

```
CaseInsensitiveString cis = new CaseInsensitiveString("Polish");
String s = "polish";
```

As expected, `cis.equals(s)` returns `true`. The problem is that while the `equals` method in `CaseInsensitiveString` knows about ordinary strings, the `equals` method in `String` is oblivious to case-insensitive strings. Therefore, `s.equals(cis)` returns `false`, a clear violation of symmetry. Suppose you put a case-insensitive string into a collection:

```
List<CaseInsensitiveString> list = new ArrayList<>();
list.add(cis);
```

What does `list.contains(s)` return at this point? Who knows? In the current OpenJDK implementation, it happens to return `false`, but that's just an implementation artifact. In another implementation, it could just as easily return `true` or throw a runtime exception. **Once you've violated the `equals` contract, you simply don't know how other objects will behave when confronted with your object.**

To eliminate the problem, merely remove the ill-conceived attempt to interoperate with `String` from the `equals` method. Once you do this, you can refactor the method into a single return statement:

```
@Override public boolean equals(Object o) {
    return o instanceof CaseInsensitiveString &&
        ((CaseInsensitiveString) o).s.equalsIgnoreCase(s);
}
```

Transitivity—The third requirement of the `equals` contract says that if one object is equal to a second and the second object is equal to a third, then the first object must be equal to the third. Again, it's not hard to imagine violating this requirement unintentionally. Consider the case of a subclass that adds a new *value component* to its superclass. In other words, the subclass adds a piece of

information that affects equals comparisons. Let's start with a simple immutable two-dimensional integer point class:

```
public class Point {
    private final int x;
    private final int y;

    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }

    @Override public boolean equals(Object o) {
        if (!(o instanceof Point))
            return false;
        Point p = (Point)o;
        return p.x == x && p.y == y;
    }

    ... // Remainder omitted
}
```

Suppose you want to extend this class, adding the notion of color to a point:

```
public class ColorPoint extends Point {
    private final Color color;

    public ColorPoint(int x, int y, Color color) {
        super(x, y);
        this.color = color;
    }

    ... // Remainder omitted
}
```

How should the equals method look? If you leave it out entirely, the implementation is inherited from Point and color information is ignored in equals comparisons. While this does not violate the equals contract, it is clearly unacceptable. Suppose you write an equals method that returns true only if its argument is another color point with the same position and color:

```
// Broken - violates symmetry!
@Override public boolean equals(Object o) {
    if (!(o instanceof ColorPoint))
        return false;
    return super.equals(o) && ((ColorPoint) o).color == color;
}
```

The problem with this method is that you might get different results when comparing a point to a color point and vice versa. The former comparison ignores color, while the latter comparison always returns `false` because the type of the argument is incorrect. To make this concrete, let's create one point and one color point:

```
Point p = new Point(1, 2);
ColorPoint cp = new ColorPoint(1, 2, Color.RED);
```

Then `p.equals(cp)` returns `true`, while `cp.equals(p)` returns `false`. You might try to fix the problem by having `ColorPoint.equals` ignore color when doing "mixed comparisons":

```
// Broken - violates transitivity!
@Override public boolean equals(Object o) {
    if (!(o instanceof Point))
        return false;

    // If o is a normal Point, do a color-blind comparison
    if (!(o instanceof ColorPoint))
        return o.equals(this);

    // o is a ColorPoint; do a full comparison
    return super.equals(o) && ((ColorPoint) o).color == color;
}
```

This approach does provide symmetry, but at the expense of transitivity:

```
ColorPoint p1 = new ColorPoint(1, 2, Color.RED);
Point p2 = new Point(1, 2);
ColorPoint p3 = new ColorPoint(1, 2, Color.BLUE);
```

Now `p1.equals(p2)` and `p2.equals(p3)` return `true`, while `p1.equals(p3)` returns `false`, a clear violation of transitivity. The first two comparisons are "color-blind," while the third takes color into account.

Also, this approach can cause infinite recursion: Suppose there are two subclasses of `Point`, say `ColorPoint` and `SmellPoint`, each with this sort of `equals` method. Then a call to `myColorPoint.equals(mySmellPoint)` will throw a `StackOverflowError`.

So what's the solution? It turns out that this is a fundamental problem of equivalence relations in object-oriented languages. **There is no way to extend an instantiable class and add a value component while preserving the `equals` contract**, unless you're willing to forgo the benefits of object-oriented abstraction.

You may hear it said that you can extend an instantiable class and add a value component while preserving the `equals` contract by using a `getClass` test in place of the `instanceof` test in the `equals` method:

```
// Broken - violates Liskov substitution principle (page 43)
@Override public boolean equals(Object o) {
    if (o == null || o.getClass() != getClass())
        return false;
    Point p = (Point) o;
    return p.x == x && p.y == y;
}
```

This has the effect of equating objects only if they have the same implementation class. This may not seem so bad, but the consequences are unacceptable: An instance of a subclass of `Point` is still a `Point`, and it still needs to function as one, but it fails to do so if you take this approach! Let's suppose we want to write a method to tell whether a point is on the unit circle. Here is one way we could do it:

```
// Initialize unitCircle to contain all Points on the unit circle
private static final Set<Point> unitCircle = Set.of(
        new Point( 1,  0), new Point( 0,  1),
        new Point(-1,  0), new Point( 0, -1));

public static boolean onUnitCircle(Point p) {
    return unitCircle.contains(p);
}
```

While this may not be the fastest way to implement the functionality, it works fine. Suppose you extend `Point` in some trivial way that doesn't add a value component, say, by having its constructor keep track of how many instances have been created:

```
public class CounterPoint extends Point {
    private static final AtomicInteger counter =
            new AtomicInteger();

    public CounterPoint(int x, int y) {
        super(x, y);
        counter.incrementAndGet();
    }
    public static int numberCreated() { return counter.get(); }
}
```

The *Liskov substitution principle* says that any important property of a type should also hold for all its subtypes so that any method written for the type should work equally well on its subtypes [Liskov87]. This is the formal statement of our

earlier claim that a subclass of `Point` (such as `CounterPoint`) is still a `Point` and must act as one. But suppose we pass a `CounterPoint` to the `onUnitCircle` method. If the `Point` class uses a `getClass`-based `equals` method, the `onUnitCircle` method will return `false` regardless of the `CounterPoint` instance's *x* and *y* coordinates. This is so because most collections, including the `HashSet` used by the `onUnitCircle` method, use the `equals` method to test for containment, and no `CounterPoint` instance is equal to any `Point`. If, however, you use a proper `instanceof`-based `equals` method on `Point`, the same `onUnitCircle` method works fine when presented with a `CounterPoint` instance.

While there is no satisfactory way to extend an instantiable class and add a value component, there is a fine workaround: Follow the advice of Item 18, "Favor composition over inheritance." Instead of having `ColorPoint` extend `Point`, give `ColorPoint` a private `Point` field and a public *view* method (Item 6) that returns the point at the same position as this color point:

```
// Adds a value component without violating the equals contract
public class ColorPoint {
   private final Point point;
   private final Color color;

   public ColorPoint(int x, int y, Color color) {
      point = new Point(x, y);
      this.color = Objects.requireNonNull(color);
   }

   /**
    * Returns the point-view of this color point.
    */
   public Point asPoint() {
      return point;
   }

   @Override public boolean equals(Object o) {
      if (!(o instanceof ColorPoint))
         return false;
      ColorPoint cp = (ColorPoint) o;
      return cp.point.equals(point) && cp.color.equals(color);
   }

   ... // Remainder omitted
}
```

There are some classes in the Java platform libraries that do extend an instantiable class and add a value component. For example, `java.sql.Timestamp`

extends `java.util.Date` and adds a `nanoseconds` field. The `equals` implementation for `Timestamp` does violate symmetry and can cause erratic behavior if `Timestamp` and `Date` objects are used in the same collection or are otherwise intermixed. The `Timestamp` class has a disclaimer cautioning programmers against mixing dates and timestamps. While you won't get into trouble as long as you keep them separate, there's nothing to prevent you from mixing them, and the resulting errors can be hard to debug. This behavior of the `Timestamp` class was a mistake and should not be emulated.

Note that you *can* add a value component to a subclass of an *abstract* class without violating the `equals` contract. This is important for the sort of class hierarchies that you get by following the advice in Item 23, "Prefer class hierarchies to tagged classes." For example, you could have an abstract class `Shape` with no value components, a subclass `Circle` that adds a `radius` field, and a subclass `Rectangle` that adds `length` and `width` fields. Problems of the sort shown earlier won't occur so long as it is impossible to create a superclass instance directly.

Consistency—The fourth requirement of the `equals` contract says that if two objects are equal, they must remain equal for all time unless one (or both) of them is modified. In other words, mutable objects can be equal to different objects at different times while immutable objects can't. When you write a class, think hard about whether it should be immutable (Item 17). If you conclude that it should, make sure that your `equals` method enforces the restriction that equal objects remain equal and unequal objects remain unequal for all time.

Whether or not a class is immutable, **do not write an `equals` method that depends on unreliable resources.** It's extremely difficult to satisfy the consistency requirement if you violate this prohibition. For example, `java.net.URL`'s `equals` method relies on comparison of the IP addresses of the hosts associated with the URLs. Translating a host name to an IP address can require network access, and it isn't guaranteed to yield the same results over time. This can cause the `URL` `equals` method to violate the `equals` contract and has caused problems in practice. The behavior of `URL`'s `equals` method was a big mistake and should not be emulated. Unfortunately, it cannot be changed due to compatibility requirements. To avoid this sort of problem, `equals` methods should perform only deterministic computations on memory-resident objects.

Non-nullity—The final requirement lacks an official name, so I have taken the liberty of calling it "non-nullity." It says that all objects must be unequal to `null`. While it is hard to imagine accidentally returning `true` in response to the invocation `o.equals(null)`, it isn't hard to imagine accidentally throwing a

`NullPointerException`. The general contract prohibits this. Many classes have `equals` methods that guard against it with an explicit test for `null`:

```
@Override public boolean equals(Object o) {
    if (o == null)
        return false;
    ...
}
```

This test is unnecessary. To test its argument for equality, the `equals` method must first cast its argument to an appropriate type so its accessors can be invoked or its fields accessed. Before doing the cast, the method must use the `instanceof` operator to check that its argument is of the correct type:

```
@Override public boolean equals(Object o) {
    if (!(o instanceof MyType))
        return false;
    MyType mt = (MyType) o;
    ...
}
```

If this type check were missing and the `equals` method were passed an argument of the wrong type, the `equals` method would throw a `ClassCastException`, which violates the `equals` contract. But the `instanceof` operator is specified to return `false` if its first operand is `null`, regardless of what type appears in the second operand [JLS, 15.20.2]. Therefore, the type check will return `false` if `null` is passed in, so you don't need an explicit `null` check.

Putting it all together, here's a recipe for a high-quality `equals` method:

1. **Use the == operator to check if the argument is a reference to this object.** If so, return `true`. This is just a performance optimization but one that is worth doing if the comparison is potentially expensive.

2. **Use the `instanceof` operator to check if the argument has the correct type.** If not, return `false`. Typically, the correct type is the class in which the method occurs. Occasionally, it is some interface implemented by this class. Use an interface if the class implements an interface that refines the `equals` contract to permit comparisons across classes that implement the interface. Collection interfaces such as `Set`, `List`, `Map`, and `Map.Entry` have this property.

3. **Cast the argument to the correct type.** Because this cast was preceded by an `instanceof` test, it is guaranteed to succeed.

4. **For each "significant" field in the class, check if that field of the argument matches the corresponding field of this object.** If all these tests succeed, return `true`; otherwise, return `false`. If the type in Step 2 is an interface, you must access the argument's fields via interface methods; if the type is a class, you may be able to access the fields directly, depending on their accessibility.

   For primitive fields whose type is not `float` or `double`, use the == operator for comparisons; for object reference fields, call the `equals` method recursively; for `float` fields, use the static `Float.compare(float, float)` method; and for `double` fields, use `Double.compare(double, double)`. The special treatment of `float` and `double` fields is made necessary by the existence of `Float.NaN`, `-0.0f` and the analogous `double` values; see JLS 15.21.1 or the documentation of `Float.equals` for details. While you could compare `float` and `double` fields with the static methods `Float.equals` and `Double.equals`, this would entail autoboxing on every comparison, which would have poor performance. For array fields, apply these guidelines to each element. If every element in an array field is significant, use one of the `Arrays.equals` methods.

   Some object reference fields may legitimately contain `null`. To avoid the possibility of a `NullPointerException`, check such fields for equality using the static method `Objects.equals(Object, Object)`.

   For some classes, such as `CaseInsensitiveString` above, field comparisons are more complex than simple equality tests. If this is the case, you may want to store a *canonical form* of the field so the `equals` method can do a cheap exact comparison on canonical forms rather than a more costly nonstandard comparison. This technique is most appropriate for immutable classes (Item 17); if the object can change, you must keep the canonical form up to date.

   The performance of the `equals` method may be affected by the order in which fields are compared. For best performance, you should first compare fields that are more likely to differ, less expensive to compare, or, ideally, both. You must not compare fields that are not part of an object's logical state, such as lock fields used to synchronize operations. You need not compare *derived fields*, which can be calculated from "significant fields," but doing so may improve the performance of the `equals` method. If a derived field amounts to a summary description of the entire object, comparing this field will save you the expense of comparing the actual data if the comparison fails. For example, suppose you have a `Polygon` class, and you cache the area. If two polygons have unequal areas, you needn't bother comparing their edges and vertices.

When you are finished writing your `equals` method, ask yourself three questions: Is it symmetric? Is it transitive? Is it consistent? And don't just ask yourself; write unit tests to check, unless you used AutoValue (page 49) to generate your `equals` method, in which case you can safely omit the tests. If the properties fail to hold, figure out why, and modify the `equals` method accordingly. Of course your `equals` method must also satisfy the other two properties (reflexivity and non-nullity), but these two usually take care of themselves.

An `equals` method constructed according to the previous recipe is shown in this simplistic `PhoneNumber` class:

```
// Class with a typical equals method
public final class PhoneNumber {
    private final short areaCode, prefix, lineNum;

    public PhoneNumber(int areaCode, int prefix, int lineNum) {
        this.areaCode = rangeCheck(areaCode, 999, "area code");
        this.prefix   = rangeCheck(prefix,   999, "prefix");
        this.lineNum  = rangeCheck(lineNum, 9999, "line num");
    }

    private static short rangeCheck(int val, int max, String arg) {
        if (val < 0 || val > max)
           throw new IllegalArgumentException(arg + ": " + val);
        return (short) val;
    }

    @Override public boolean equals(Object o) {
        if (o == this)
            return true;
        if (!(o instanceof PhoneNumber))
            return false;
        PhoneNumber pn = (PhoneNumber)o;
        return pn.lineNum == lineNum && pn.prefix == prefix
                && pn.areaCode == areaCode;
    }
    ... // Remainder omitted
}
```

Here are a few final caveats:

- **Always override `hashCode` when you override `equals`** (Item 11).
- **Don't try to be too clever.** If you simply test fields for equality, it's not hard to adhere to the `equals` contract. If you are overly aggressive in searching for equivalence, it's easy to get into trouble. It is generally a bad idea to take any form of aliasing into account. For example, the `File` class shouldn't attempt to equate symbolic links referring to the same file. Thankfully, it doesn't.

- **Don't substitute another type for `Object` in the `equals` declaration.** It is not uncommon for a programmer to write an `equals` method that looks like this and then spend hours puzzling over why it doesn't work properly:

```
// Broken - parameter type must be Object!
public boolean equals(MyClass o) {
    ...
}
```

The problem is that this method does not *override* `Object.equals`, whose argument is of type `Object`, but *overloads* it instead (Item 52). It is unacceptable to provide such a "strongly typed" `equals` method even in addition to the normal one, because it can cause `Override` annotations in subclasses to generate false positives and provide a false sense of security.

Consistent use of the `Override` annotation, as illustrated throughout this item, will prevent you from making this mistake (Item 40). This `equals` method won't compile, and the error message will tell you exactly what is wrong:

```
// Still broken, but won't compile
@Override public boolean equals(MyClass o) {
    ...
}
```

Writing and testing `equals` (and `hashCode`) methods is tedious, and the resulting code is mundane. An excellent alternative to writing and testing these methods manually is to use Google's open source AutoValue framework, which automatically generates these methods for you, triggered by a single annotation on the class . In most cases, the methods generated by AutoValue are essentially identical to those you'd write yourself.

IDEs, too, have facilities to generate `equals` and `hashCode` methods, but the resulting source code is more verbose and less readable than code that uses AutoValue, does not track changes in the class automatically, and therefore requires testing. That said, having IDEs generate `equals` (and `hashCode`) methods is generally preferable to implementing them manually because IDEs do not make careless mistakes, and humans do.

In summary, don't override the `equals` method unless you have to: in many cases, the implementation inherited from `Object` does exactly what you want. If you do override `equals`, make sure to compare all of the class's significant fields and to compare them in a manner that preserves all five provisions of the `equals` contract.