**A**LTHOUGH 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](https://learning.oreilly.com/library/view/effective-java-3rd/9780134686097/ch2.xhtml#lev8). While not an Object method, Comparable.compareTo is discussed in this chapter because it has a similar character.

**Obey the general contract when overriding equals**

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:

Code snippet of equals function from Objects class.

public static boolean equals(Object a, Object b) {  
 return a == b || a != null && a.equals(b);  
}

• **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:

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.

### ****Always override**** ****hashCode**** ****when you override**** ****equals****

If you fail to do so, your class will violate the general contract for hashCode, which will prevent it from functioning properly in collections such as HashMap and HashSet. Here is the contract, adapted from the Object specification :

• When the hashCode method is invoked on an object repeatedly during an execution of an application, it must consistently return the same value, provided no information used in equals comparisons is modified. This value need not remain consistent from one execution of an application to another.

• If two objects are equal according to the equals(Object) method, then calling hashCode on the two objects must produce the same integer result.

• If two objects are unequal according to the equals(Object) method, it is *not* required that calling hashCode on each of the objects must produce distinct results. However, the programmer should be aware that producing distinct results for unequal objects may improve the performance of hash tables.

**The key provision that is violated when you fail to override** **hashCode** **is the second one: equal objects must have equal hash codes.** Two distinct instances may be logically equal according to a class’s equals method, but to Object’s hashCode method, they’re just two objects with nothing much in common. Therefore, Object’s hashCode method returns two seemingly random numbers instead of two equal numbers as required by the contract.

Map<PhoneNumber, String> m = new HashMap<>();

m.put(new PhoneNumber(707, 867, 5309), "Jenny");

At this point, you might expect m.get(new PhoneNumber(707, 867, 5309)) to return "Jenny", but instead, it returns null. Notice that two PhoneNumber instances are involved: one is used for insertion into the HashMap, and a second, equal instance is used for (attempted) retrieval. The PhoneNumber class’s failure to override hashCode causes the two equal instances to have unequal hash codes, in violation of the hashCode contract. Therefore, the get method is likely to look for the phone number in a different hash bucket from the one in which it was stored by the put method. Even if the two instances happen to hash to the same bucket, the get method will almost certainly return null, because HashMap has an optimization that caches the hash code associated with each entry and doesn’t bother checking for object equality if the hash codes don’t match.

A good hash function tends to produce unequal hash codes for unequal instances. The Objects class has a static method that takes an arbitrary number of objects and returns a hash code for them. This method, named hash, lets you write one-line hashCode methods whose quality is comparable to those written according to the recipe in this item. Unfortunately, they run more slowly because they entail array creation to pass a variable number of arguments, as well as boxing and unboxing if any of the arguments are of primitive type.

Do not be tempted to exclude significant fields from the hash code computation to improve performance.

In summary, you *must* override hashCode every time you override equals, or your program will not run correctly. Your hashCode method must obey the general contract specified in Object and must do a reasonable job assigning unequal hash codes to unequal instances. This is easy to achieve, if slightly tedious, using the recipe on page 51. As mentioned in [Item 10](https://learning.oreilly.com/library/view/effective-java-3rd/9780134686097/ch3.xhtml#lev10), the AutoValue framework provides a fine alternative to writing equals and hashCode methods manually, and IDEs also provide some of this functionality.

### ****Always override**** ****toString****

While Object provides an implementation of the toString method, the string that it returns is generally not what the user of your class wants to see. It consists of the class name followed by an “at” sign (@) and the unsigned hexadecimal representation of the hash code, for example, PhoneNumber@adbbd. When practical, the toString method should return all of the interesting information contained in the object.

To recap, override Object’s toString implementation in every instantiable class you write, unless a superclass has already done so. It makes classes much more pleasant to use and aids in debugging. The toString method should return a concise, useful description of the object, in an aesthetically pleasing format

### ****Override**** ****clone**** ****judiciously****

So what *does* Cloneable do, given that it contains no methods? It determines the behavior of Object’s protected clone implementation: if a class implements Cloneable, Object’s clone method returns a field-by-field copy of the object; otherwise it throws CloneNotSupportedException. This is a highly atypical use of interfaces and not one to be emulated. Normally, implementing an interface says something about what a class can do for its clients. In this case, it modifies the behavior of a protected method on a superclass. In effect, the clone method functions as a constructor; you must ensure that it does no harm to the original object and that it properly establishes invariants on the clone

To recap, all classes that implement Cloneable should override clone with a public method whose return type is the class itself. This method should first call super.clone, then fix any fields that need fixing. Typically, this means copying any mutable objects that comprise the internal “deep structure” of the object and replacing the clone’s references to these objects with references to their copies. While these internal copies can usually be made by calling clone recursively, this is not always the best approach. If the class contains only primitive fields or references to immutable objects, then it is likely the case that no fields need to be fixed. There are exceptions to this rule. For example, a field representing a serial number or other unique ID will need to be fixed even if it is primitive or immutable.

### ****Consider implementing**** ****Comparable****

Unlike the other methods discussed in this chapter, the compareTo method is not declared in Object. Rather, it is the sole method in the Comparable interface. It is similar in character to Object’s equals method, except that it permits order comparisons in addition to simple equality comparisons, and it is generic. By implementing Comparable, a class indicates that its instances have a *natural ordering.* Sorting an array of objects that implement Comparable is as simple as this:

Arrays.sort(a);

In the following description, the notation sgn(*expression*) designates the mathematical *signum* function, which is defined to return -1, 0, or 1, according to whether the value of *expression* is negative, zero, or positive.

• The implementor must ensure that sgn(x.compareTo(y)) == -sgn(y. compareTo(x)) for all x and y. (This implies that x.compareTo(y) must throw an exception if and only if y.compareTo(x) throws an exception.)

• The implementor must also ensure that the relation is transitive: (x. compareTo(y) > 0 && y.compareTo(z) > 0) implies x.compareTo(z) > 0.

• Finally, the implementor must ensure that x.compareTo(y) == 0 implies that sgn(x.compareTo(z)) == sgn(y.compareTo(z)), for all z.

• It is strongly recommended, but not required, that (x.compareTo(y) == 0) == (x.equals(y)). Generally speaking, any class that implements the Comparable interface and violates this condition should clearly indicate this fact. The recommended language is “Note: This class has a natural ordering that is inconsistent with equals.”

In summary, whenever you implement a value class that has a sensible ordering, you should have the class implement the Comparable interface so that its instances can be easily sorted, searched, and used in comparison-based collections. When comparing field values in the implementations of the compareTo methods, avoid the use of the < and > operators. Instead, use the static compare methods in the boxed primitive classes or the comparator construction methods in the Comparator interface.