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.

Item 11: Always override hashCode when you override equals

You must override hashCode in every class that overrides 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.

For example, suppose you attempt to use instances of the PhoneNumber class from Item 10 as keys in a HashMap:

```
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.

Fixing this problem is as simple as writing a proper `hashCode` method for `PhoneNumber`. So what should a `hashCode` method look like? It's trivial to write a bad one. This one, for example, is always legal but should never be used:

```
// The worst possible legal hashCode implementation - never use!
@Override public int hashCode() { return 42; }
```

It's legal because it ensures that equal objects have the same hash code. It's atrocious because it ensures that *every* object has the same hash code. Therefore, every object hashes to the same bucket, and hash tables degenerate to linked lists. Programs that should run in linear time instead run in quadratic time. For large hash tables, this is the difference between working and not working.

A good hash function tends to produce unequal hash codes for unequal instances. This is exactly what is meant by the third part of the `hashCode` contract. Ideally, a hash function should distribute any reasonable collection of unequal instances uniformly across all `int` values. Achieving this ideal can be difficult. Luckily it's not too hard to achieve a fair approximation. Here is a simple recipe:

1. Declare an `int` variable named `result`, and initialize it to the hash code `c` for the first significant field in your object, as computed in step 2.a. (Recall from Item 10 that a significant field is a field that affects equals comparisons.)
2. For every remaining significant field `f` in your object, do the following:
   a. Compute an `int` hash code `c` for the field:
      i. If the field is of a primitive type, compute *`Type`*`.hashCode(f)`, where *`Type`* is the boxed primitive class corresponding to `f`'s type.
      ii. If the field is an object reference and this class's `equals` method compares the field by recursively invoking `equals`, recursively invoke `hashCode` on the field. If a more complex comparison is required, compute a "canonical representation" for this field and invoke `hashCode` on the canonical representation. If the value of the field is `null`, use `0` (or some other constant, but `0` is traditional).
      iii. If the field is an array, treat it as if each significant element were a separate field. That is, compute a hash code for each significant element by applying these rules recursively, and combine the values per step 2.b. If the array has no significant elements, use a constant, preferably not `0`. If all elements are significant, use `Arrays.hashCode`.
   b. Combine the hash code `c` computed in step 2.a into `result` as follows:
      ```
      result = 31 * result + c;
      ```
3. Return `result`.

When you are finished writing the `hashCode` method, ask yourself whether equal instances have equal hash codes. Write unit tests to verify your intuition (unless you used AutoValue to generate your `equals` and `hashCode` methods, in which case you can safely omit these tests). If equal instances have unequal hash codes, figure out why and fix the problem.

You may exclude *derived fields* from the hash code computation. In other words, you may ignore any field whose value can be computed from fields included in the computation. You *must* exclude any fields that are not used in `equals` comparisons, or you risk violating the second provision of the `hashCode` contract.

The multiplication in step 2.b makes the result depend on the order of the fields, yielding a much better hash function if the class has multiple similar fields. For example, if the multiplication were omitted from a `String` hash function, all anagrams would have identical hash codes. The value 31 was chosen because it is an odd prime. If it were even and the multiplication overflowed, information would be lost, because multiplication by 2 is equivalent to shifting. The advantage of using a prime is less clear, but it is traditional. A nice property of 31 is that the multiplication can be replaced by a shift and a subtraction for better performance on some architectures: `31 * i == (i << 5) - i`. Modern VMs do this sort of optimization automatically.

Let's apply the previous recipe to the `PhoneNumber` class:

```
// Typical hashCode method
@Override public int hashCode() {
    int result = Short.hashCode(areaCode);
    result = 31 * result + Short.hashCode(prefix);
    result = 31 * result + Short.hashCode(lineNum);
    return result;
}
```

Because this method returns the result of a simple deterministic computation whose only inputs are the three significant fields in a `PhoneNumber` instance, it is clear that equal `PhoneNumber` instances have equal hash codes. This method is, in fact, a perfectly good `hashCode` implementation for `PhoneNumber`, on par with those in the Java platform libraries. It is simple, is reasonably fast, and does a reasonable job of dispersing unequal phone numbers into different hash buckets.

While the recipe in this item yields reasonably good hash functions, they are not state-of-the-art. They are comparable in quality to the hash functions found in the Java platform libraries' value types and are adequate for most uses. If you have a bona fide need for hash functions less likely to produce collisions, see Guava's `com.google.common.hash.Hashing` [Guava].

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. This style of hash function is recommended for use only in situations where performance is not critical. Here is a hash function for `PhoneNumber` written using this technique:

```
// One-line hashCode method - mediocre performance
@Override public int hashCode() {
   return Objects.hash(lineNum, prefix, areaCode);
}
```

If a class is immutable and the cost of computing the hash code is significant, you might consider caching the hash code in the object rather than recalculating it each time it is requested. If you believe that most objects of this type will be used as hash keys, then you should calculate the hash code when the instance is created. Otherwise, you might choose to *lazily initialize* the hash code the first time `hashCode` is invoked. Some care is required to ensure that the class remains thread-safe in the presence of a lazily initialized field (Item 83). Our `PhoneNumber` class does not merit this treatment, but just to show you how it's done, here it is. Note that the initial value for the `hashCode` field (in this case, 0) should not be the hash code of a commonly created instance:

```
// hashCode method with lazily initialized cached hash code
private int hashCode; // Automatically initialized to 0

@Override public int hashCode() {
    int result = hashCode;
    if (result == 0) {
        result = Short.hashCode(areaCode);
        result = 31 * result + Short.hashCode(prefix);
        result = 31 * result + Short.hashCode(lineNum);
        hashCode = result;
    }
    return result;
}
```

Do not be tempted to exclude significant fields from the hash code computation to improve performance. While the resulting hash function may run faster, its poor quality may degrade hash tables' performance to the point where they become unusable. In particular, the hash function may be confronted with a

large collection of instances that differ mainly in regions you've chosen to ignore. If this happens, the hash function will map all these instances to a few hash codes, and programs that should run in linear time will instead run in quadratic time.

This is not just a theoretical problem. Prior to Java 2, the `String` hash function used at most sixteen characters evenly spaced throughout the string, starting with the first character. For large collections of hierarchical names, such as URLs, this function displayed exactly the pathological behavior described earlier.

Don't provide a detailed specification for the value returned by `hashCode`, so clients can't reasonably depend on it; this gives you the flexibility to change it. Many classes in the Java libraries, such as `String` and `Integer`, specify the exact value returned by their `hashCode` method as a function of the instance value. This is *not* a good idea but a mistake that we're forced to live with: It impedes the ability to improve the hash function in future releases. If you leave the details unspecified and a flaw is found in the hash function or a better hash function is discovered, you can change it in a subsequent release.

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, the AutoValue framework provides a fine alternative to writing `equals` and `hashCode` methods manually, and IDEs also provide some of this functionality.

Item 12: 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@163b91`. The general contract for `toString` says that the returned string should be "a concise but informative representation that is easy for a person to read." While it could be argued that `PhoneNumber@163b91` is concise and easy to read, it isn't very informative when compared to `707-867-5309`. The `toString` contract goes on to say, "It is recommended that all subclasses override this method." Good advice, indeed!

While it isn't as critical as obeying the `equals` and `hashCode` contracts (Items 10 and 11), **providing a good `toString` implementation makes your class much more pleasant to use and makes systems using the class easier to debug**. The `toString` method is automatically invoked when an object is passed to `println`, `printf`, the string concatenation operator, or `assert`, or is printed by a debugger. Even if you never call `toString` on an object, others may. For example, a component that has a reference to your object may include the string representation of the object in a logged error message. If you fail to override `toString`, the message may be all but useless.

If you've provided a good `toString` method for `PhoneNumber`, generating a useful diagnostic message is as easy as this:

```
System.out.println("Failed to connect to " + phoneNumber);
```

Programmers will generate diagnostic messages in this fashion whether or not you override `toString`, but the messages won't be useful unless you do. The benefits of providing a good `toString` method extend beyond instances of the class to objects containing references to these instances, especially collections. Which would you rather see when printing a map, `{Jenny=PhoneNumber@163b91}` or `{Jenny=707-867-5309}`?

When practical, the `toString` method should return *all* of the interesting information contained in the object, as shown in the phone number example. It is impractical if the object is large or if it contains state that is not conducive to string representation. Under these circumstances, `toString` should return a summary such as `Manhattan residential phone directory (1487536 listings)` or `Thread[main,5,main]`. Ideally, the string should be self-explanatory. (The `Thread` example flunks this test.) A particularly annoying penalty for failing to

include all of an object's interesting information in its string representation is test failure reports that look like this:

```
Assertion failure: expected {abc, 123}, but was {abc, 123}.
```

One important decision you'll have to make when implementing a `toString` method is whether to specify the format of the return value in the documentation. It is recommended that you do this for *value classes*, such as phone number or matrix. The advantage of specifying the format is that it serves as a standard, unambiguous, human-readable representation of the object. This representation can be used for input and output and in persistent human-readable data objects, such as CSV files. If you specify the format, it's usually a good idea to provide a matching static factory or constructor so programmers can easily translate back and forth between the object and its string representation. This approach is taken by many value classes in the Java platform libraries, including `BigInteger`, `BigDecimal`, and most of the boxed primitive classes.

The disadvantage of specifying the format of the `toString` return value is that once you've specified it, you're stuck with it for life, assuming your class is widely used. Programmers will write code to parse the representation, to generate it, and to embed it into persistent data. If you change the representation in a future release, you'll break their code and data, and they will yowl. By choosing not to specify a format, you preserve the flexibility to add information or improve the format in a subsequent release.

Whether or not you decide to specify the format, you should clearly document your intentions. If you specify the format, you should do so precisely. For example, here's a `toString` method to go with the `PhoneNumber` class in Item 11:

```
/**
 * Returns the string representation of this phone number.
 * The string consists of twelve characters whose format is
 * "XXX-YYY-ZZZZ", where XXX is the area code, YYY is the
 * prefix, and ZZZZ is the line number. Each of the capital
 * letters represents a single decimal digit.
 *
 * If any of the three parts of this phone number is too small
 * to fill up its field, the field is padded with leading zeros.
 * For example, if the value of the line number is 123, the last
 * four characters of the string representation will be "0123".
 */
@Override public String toString() {
    return String.format("%03d-%03d-%04d",
            areaCode, prefix, lineNum);
}
```

If you decide not to specify a format, the documentation comment should read something like this:

```
/**
 * Returns a brief description of this potion. The exact details
 * of the representation are unspecified and subject to change,
 * but the following may be regarded as typical:
 *
 * "[Potion #9: type=love, smell=turpentine, look=india ink]"
 */
@Override public String toString() { ... }
```

After reading this comment, programmers who produce code or persistent data that depends on the details of the format will have no one but themselves to blame when the format is changed.

Whether or not you specify the format, **provide programmatic access to the information contained in the value returned by `toString`.** For example, the `PhoneNumber` class should contain accessors for the area code, prefix, and line number. If you fail to do this, you *force* programmers who need this information to parse the string. Besides reducing performance and making unnecessary work for programmers, this process is error-prone and results in fragile systems that break if you change the format. By failing to provide accessors, you turn the string format into a de facto API, even if you've specified that it's subject to change.

It makes no sense to write a `toString` method in a static utility class (Item 4). Nor should you write a `toString` method in most enum types (Item 34) because Java provides a perfectly good one for you. You should, however, write a `toString` method in any abstract class whose subclasses share a common string representation. For example, the `toString` methods on most collection implementations are inherited from the abstract collection classes.

Google's open source AutoValue facility, discussed in Item 10, will generate a `toString` method for you, as will most IDEs. These methods are great for telling you the contents of each field but aren't specialized to the *meaning* of the class. So, for example, it would be inappropriate to use an automatically generated `toString` method for our `PhoneNumber` class (as phone numbers have a standard string representation), but it would be perfectly acceptable for our `Potion` class. That said, an automatically generated `toString` method is far preferable to the one inherited from `Object`, which tells you *nothing* about an object's value.

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.

Item 13: Override `clone` judiciously

The `Cloneable` interface was intended as a *mixin interface* (Item 20) for classes to advertise that they permit cloning. Unfortunately, it fails to serve this purpose. Its primary flaw is that it lacks a `clone` method, and `Object`'s `clone` method is protected. You cannot, without resorting to *reflection* (Item 65), invoke `clone` on an object merely because it implements `Cloneable`. Even a reflective invocation may fail, because there is no guarantee that the object has an accessible `clone` method. Despite this flaw and many others, the facility is in reasonably wide use, so it pays to understand it. This item tells you how to implement a well-behaved `clone` method, discusses when it is appropriate to do so, and presents alternatives.

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.

Though the specification doesn't say it, **in practice, a class implementing `Cloneable` is expected to provide a properly functioning public `clone` method.** In order to achieve this, the class and all of its superclasses must obey a complex, unenforceable, thinly documented protocol. The resulting mechanism is fragile, dangerous, and *extralinguistic*: it creates objects without calling a constructor.

The general contract for the `clone` method is weak. Here it is, copied from the `Object` specification :

> Creates and returns a copy of this object. The precise meaning of "copy" may depend on the class of the object. The general intent is that, for any object `x`, the expression
>
> ```
> x.clone() != x
> ```
>
> will be `true`, and the expression
>
> ```
> x.clone().getClass() == x.getClass()
> ```
>
> will be `true`, but these are not absolute requirements. While it is typically the case that
>
> ```
> x.clone().equals(x)
> ```
>
> will be `true`, this is not an absolute requirement.

By convention, the object returned by this method should be obtained by calling `super.clone`. If a class and all of its superclasses (except `Object`) obey this convention, it will be the case that

`x.clone().getClass() == x.getClass()`.

By convention, the returned object should be independent of the object being cloned. To achieve this independence, it may be necessary to modify one or more fields of the object returned by `super.clone` before returning it.

This mechanism is vaguely similar to constructor chaining, except that it isn't enforced: if a class's `clone` method returns an instance that is *not* obtained by calling `super.clone` but by calling a constructor, the compiler won't complain, but if a subclass of that class calls `super.clone`, the resulting object will have the wrong class, preventing the subclass from `clone` method from working properly. If a class that overrides `clone` is final, this convention may be safely ignored, as there are no subclasses to worry about. But if a final class has a `clone` method that does not invoke `super.clone`, there is no reason for the class to implement `Cloneable`, as it doesn't rely on the behavior of `Object`'s clone implementation.

Suppose you want to implement `Cloneable` in a class whose superclass provides a well-behaved `clone` method. First call `super.clone`. The object you get back will be a fully functional replica of the original. Any fields declared in your class will have values identical to those of the original. If every field contains a primitive value or a reference to an immutable object, the returned object may be exactly what you need, in which case no further processing is necessary. This is the case, for example, for the `PhoneNumber` class in Item 11, but note that **immutable classes should never provide a `clone` method** because it would merely encourage wasteful copying. With that caveat, here's how a `clone` method for `PhoneNumber` would look:

```
// Clone method for class with no references to mutable state
@Override public PhoneNumber clone() {
    try {
        return (PhoneNumber) super.clone();
    } catch (CloneNotSupportedException e) {
        throw new AssertionError();  // Can't happen
    }
}
```

In order for this method to work, the class declaration for `PhoneNumber` would have to be modified to indicate that it implements `Cloneable`. Though `Object`'s `clone` method returns `Object`, this `clone` method returns `PhoneNumber`. It is legal

and desirable to do this because Java supports *covariant return types*. In other words, an overriding method's return type can be a subclass of the overridden method's return type. This eliminates the need for casting in the client. We must cast the result of `super.clone` from `Object` to `PhoneNumber` before returning it, but the cast is guaranteed to succeed.

The call to `super.clone` is contained in a `try-catch` block. This is because `Object` declares its `clone` method to throw `CloneNotSupportedException`, which is a *checked exception*. Because `PhoneNumber` implements `Cloneable`, we know the call to `super.clone` will succeed. The need for this boilerplate indicates that `CloneNotSupportedException` should have been unchecked (Item 71).

If an object contains fields that refer to mutable objects, the simple `clone` implementation shown earlier can be disastrous. For example, consider the `Stack` class in Item 7:

```
public class Stack {
    private Object[] elements;
    private int size = 0;
    private static final int DEFAULT_INITIAL_CAPACITY = 16;

    public Stack() {
        this.elements = new Object[DEFAULT_INITIAL_CAPACITY];
    }

    public void push(Object e) {
        ensureCapacity();
        elements[size++] = e;
    }

    public Object pop() {
        if (size == 0)
            throw new EmptyStackException();
        Object result = elements[--size];
        elements[size] = null; // Eliminate obsolete reference
        return result;
    }

    // Ensure space for at least one more element.
    private void ensureCapacity() {
        if (elements.length == size)
            elements = Arrays.copyOf(elements, 2 * size + 1);
    }
}
```

Suppose you want to make this class cloneable. If the `clone` method merely returns `super.clone()`, the resulting `Stack` instance will have the correct value in

its `size` field, but its `elements` field will refer to the same array as the original `Stack` instance. Modifying the original will destroy the invariants in the clone and vice versa. You will quickly find that your program produces nonsensical results or throws a `NullPointerException`.

This situation could never occur as a result of calling the sole constructor in the `Stack` class. **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**. In order for the `clone` method on `Stack` to work properly, it must copy the internals of the stack. The easiest way to do this is to call `clone` recursively on the `elements` array:

```
// Clone method for class with references to mutable state
@Override public Stack clone() {
    try {
        Stack result = (Stack) super.clone();
        result.elements = elements.clone();
        return result;
    } catch (CloneNotSupportedException e) {
        throw new AssertionError();
    }
}
```

Note that we do not have to cast the result of `elements.clone` to `Object[]`. Calling `clone` on an array returns an array whose runtime and compile-time types are identical to those of the array being cloned. This is the preferred idiom to duplicate an array. In fact, arrays are the sole compelling use of the `clone` facility.

Note also that the earlier solution would not work if the `elements` field were final because `clone` would be prohibited from assigning a new value to the field. This is a fundamental problem: like serialization, **the `Cloneable` architecture is incompatible with normal use of final fields referring to mutable objects**, except in cases where the mutable objects may be safely shared between an object and its clone. In order to make a class cloneable, it may be necessary to remove `final` modifiers from some fields.

It is not always sufficient merely to call `clone` recursively. For example, suppose you are writing a `clone` method for a hash table whose internals consist of an array of buckets, each of which references the first entry in a linked list of key-value pairs. For performance, the class implements its own lightweight singly linked list instead of using `java.util.LinkedList` internally:

```
public class HashTable implements Cloneable {
    private Entry[] buckets = ...;
```

```
    private static class Entry {
        final Object key;
        Object value;
        Entry  next;

        Entry(Object key, Object value, Entry next) {
            this.key   = key;
            this.value = value;
            this.next  = next;
        }
    }
    ... // Remainder omitted
}
```

Suppose you merely clone the bucket array recursively, as we did for `Stack`:

```
// Broken clone method - results in shared mutable state!
@Override public HashTable clone() {
    try {
        HashTable result = (HashTable) super.clone();
        result.buckets = buckets.clone();
        return result;
    } catch (CloneNotSupportedException e) {
        throw new AssertionError();
    }
}
```

Though the clone has its own bucket array, this array references the same linked lists as the original, which can easily cause nondeterministic behavior in both the clone and the original. To fix this problem, you'll have to copy the linked list that comprises each bucket. Here is one common approach:

```
// Recursive clone method for class with complex mutable state
public class HashTable implements Cloneable {
    private Entry[] buckets = ...;

    private static class Entry {
        final Object key;
        Object value;
        Entry  next;

        Entry(Object key, Object value, Entry next) {
            this.key   = key;
            this.value = value;
            this.next  = next;
        }
```

```
        // Recursively copy the linked list headed by this Entry
        Entry deepCopy() {
            return new Entry(key, value,
                next == null ? null : next.deepCopy());
        }
    }

    @Override public HashTable clone() {
        try {
            HashTable result = (HashTable) super.clone();
            result.buckets = new Entry[buckets.length];
            for (int i = 0; i < buckets.length; i++)
                if (buckets[i] != null)
                    result.buckets[i] = buckets[i].deepCopy();
            return result;
        } catch (CloneNotSupportedException e) {
            throw new AssertionError();
        }
    }
    ... // Remainder omitted
}
```

The private class `HashTable.Entry` has been augmented to support a "deep copy" method. The `clone` method on `HashTable` allocates a new `buckets` array of the proper size and iterates over the original `buckets` array, deep-copying each nonempty bucket. The `deepCopy` method on `Entry` invokes itself recursively to copy the entire linked list headed by the entry. While this technique is cute and works fine if the buckets aren't too long, it is not a good way to clone a linked list because it consumes one stack frame for each element in the list. If the list is long, this could easily cause a stack overflow. To prevent this from happening, you can replace the recursion in `deepCopy` with iteration:

```
// Iteratively copy the linked list headed by this Entry
Entry deepCopy() {
   Entry result = new Entry(key, value, next);
   for (Entry p = result; p.next != null; p = p.next)
      p.next = new Entry(p.next.key, p.next.value, p.next.next);
   return result;
}
```

A final approach to cloning complex mutable objects is to call `super.clone`, set all of the fields in the resulting object to their initial state, and then call higher-level methods to regenerate the state of the original object. In the case of our `HashTable` example, the `buckets` field would be initialized to a new bucket array, and the `put(key, value)` method (not shown) would be invoked for each key-

value mapping in the hash table being cloned. This approach typically yields a simple, reasonably elegant `clone` method that does not run as quickly as one that directly manipulates the innards of the clone. While this approach is clean, it is antithetical to the whole `Cloneable` architecture because it blindly overwrites the field-by-field object copy that forms the basis of the architecture.

Like a constructor, a `clone` method must never invoke an overridable method on the clone under construction (Item 19). If `clone` invokes a method that is overridden in a subclass, this method will execute before the subclass has had a chance to fix its state in the clone, quite possibly leading to corruption in the clone and the original. Therefore, the `put(key, value)` method discussed in the previous paragraph should be either final or private. (If it is private, it is presumably the "helper method" for a nonfinal public method.)

`Object`'s `clone` method is declared to throw `CloneNotSupportedException`, but overriding methods need not. **Public `clone` methods should omit the `throws` clause**, as methods that don't throw checked exceptions are easier to use (Item 71).

You have two choices when designing a class for inheritance (Item 19), but whichever one you choose, the class should *not* implement `Cloneable`. You may choose to mimic the behavior of `Object` by implementing a properly functioning protected `clone` method that is declared to throw `CloneNotSupportedException`. This gives subclasses the freedom to implement `Cloneable` or not, just as if they extended `Object` directly. Alternatively, you may choose *not* to implement a working `clone` method, and to prevent subclasses from implementing one, by providing the following degenerate `clone` implementation:

```
// clone method for extendable class not supporting Cloneable
@Override
protected final Object clone() throws CloneNotSupportedException {
    throw new CloneNotSupportedException();
}
```

There is one more detail that bears noting. If you write a thread-safe class that implements `Cloneable`, remember that its `clone` method must be properly synchronized, just like any other method (Item 78). `Object`'s `clone` method is not synchronized, so even if its implementation is otherwise satisfactory, you may have to write a synchronized `clone` method that returns `super.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.

Is all this complexity really necessary? Rarely. If you extend a class that already implements `Cloneable`, you have little choice but to implement a well-behaved `clone` method. Otherwise, you are usually better off providing an alternative means of object copying. **A better approach to object copying is to provide a *copy constructor*** or ***copy factory.*** A copy constructor is simply a constructor that takes a single argument whose type is the class containing the constructor, for example,

```
// Copy constructor
public Yum(Yum yum) { ... };
```

A copy factory is the static factory (Item 1) analogue of a copy constructor:

```
// Copy factory
public static Yum newInstance(Yum yum) { ... };
```

The copy constructor approach and its static factory variant have many advantages over `Cloneable`/`clone`: they don't rely on a risk-prone extralinguistic object creation mechanism; they don't demand unenforceable adherence to thinly documented conventions; they don't conflict with the proper use of final fields; they don't throw unnecessary checked exceptions; and they don't require casts.

Furthermore, a copy constructor or factory can take an argument whose type is an interface implemented by the class. For example, by convention all general-purpose collection implementations provide a constructor whose argument is of type `Collection` or `Map`. Interface-based copy constructors and factories, more properly known as *conversion constructors* and *conversion factories*, allow the client to choose the implementation type of the copy rather than forcing the client to accept the implementation type of the original. For example, suppose you have a `HashSet`, `s`, and you want to copy it as a `TreeSet`. The `clone` method can't offer this functionality, but it's easy with a conversion constructor: `new TreeSet<>(s)`.

Given all the problems associated with `Cloneable`, new interfaces should not extend it, and new extendable classes should not implement it. While it's less harmful for final classes to implement `Cloneable`, this should be viewed as a performance optimization, reserved for the rare cases where it is justified (Item 67). As a rule, copy functionality is best provided by constructors or factories. A notable exception to this rule is arrays, which are best copied with the clone method.

Item 14: 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);
```

It is similarly easy to search, compute extreme values, and maintain automatically sorted collections of `Comparable` objects. For example, the following program, which relies on the fact that `String` implements `Comparable`, prints an alphabetized list of its command-line arguments with duplicates eliminated:

```
public class WordList {
    public static void main(String[] args) {
        Set<String> s = new TreeSet<>();
        Collections.addAll(s, args);
        System.out.println(s);
    }
}
```

By implementing `Comparable`, you allow your class to interoperate with all of the many generic algorithms and collection implementations that depend on this interface. You gain a tremendous amount of power for a small amount of effort. Virtually all of the value classes in the Java platform libraries, as well as all enum types (Item 34), implement `Comparable`. If you are writing a value class with an obvious natural ordering, such as alphabetical order, numerical order, or chronological order, you should implement the `Comparable` interface:

```
public interface Comparable<T> {
    int compareTo(T t);
}
```

The general contract of the `compareTo` method is similar to that of `equals`:

> Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object. Throws `ClassCastException` if the specified object's type prevents it from being compared to this object.

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`."

Don't be put off by the mathematical nature of this contract. Like the `equals` contract (Item 10), this contract isn't as complicated as it looks. Unlike the `equals` method, which imposes a global equivalence relation on all objects, `compareTo` doesn't have to work across objects of different types: when confronted with objects of different types, `compareTo` is permitted to throw `ClassCastException`. Usually, that is exactly what it does. The contract does *permit* intertype comparisons, which are typically defined in an interface implemented by the objects being compared.

Just as a class that violates the `hashCode` contract can break other classes that depend on hashing, a class that violates the `compareTo` contract can break other classes that depend on comparison. Classes that depend on comparison include the sorted collections `TreeSet` and `TreeMap` and the utility classes `Collections` and `Arrays`, which contain searching and sorting algorithms.

Let's go over the provisions of the `compareTo` contract. The first provision says that if you reverse the direction of a comparison between two object references, the expected thing happens: if the first object is less than the second, then the second must be greater than the first; if the first object is equal to the second, then the second must be equal to the first; and if the first object is greater than the second, then the second must be less than the first. The second provision says that if one object is greater than a second and the second is greater than a third, then the first must be greater than the third. The final provision says that all objects that compare as equal must yield the same results when compared to any other object.

One consequence of these three provisions is that the equality test imposed by a `compareTo` method must obey the same restrictions imposed by the `equals` contract: reflexivity, symmetry, and transitivity. Therefore, the same caveat applies: there is no way to extend an instantiable class with a new value component while preserving the `compareTo` contract, unless you are willing to forgo the benefits of object-oriented abstraction (Item 10). The same workaround applies, too. If you want to add a value component to a class that implements `Comparable`, don't extend it; write an unrelated class containing an instance of the first class. Then provide a "view" method that returns the contained instance. This frees you to implement whatever `compareTo` method you like on the containing class, while allowing its client to view an instance of the containing class as an instance of the contained class when needed.

The final paragraph of the `compareTo` contract, which is a strong suggestion rather than a true requirement, simply states that the equality test imposed by the `compareTo` method should generally return the same results as the `equals` method. If this provision is obeyed, the ordering imposed by the `compareTo` method is said to be *consistent with `equals`*. If it's violated, the ordering is said to be *inconsistent with `equals`*. A class whose `compareTo` method imposes an order that is inconsistent with `equals` will still work, but sorted collections containing elements of the class may not obey the general contract of the appropriate collection interfaces (`Collection`, `Set`, or `Map`). This is because the general contracts for these interfaces are defined in terms of the `equals` method, but sorted collections use the equality test imposed by `compareTo` in place of `equals`. It is not a catastrophe if this happens, but it's something to be aware of.

For example, consider the `BigDecimal` class, whose `compareTo` method is inconsistent with `equals`. If you create an empty `HashSet` instance and then add `new BigDecimal("1.0")` and `new BigDecimal("1.00")`, the set will contain two elements because the two `BigDecimal` instances added to the set are unequal when compared using the `equals` method. If, however, you perform the same procedure using a `TreeSet` instead of a `HashSet`, the set will contain only one element because the two `BigDecimal` instances are equal when compared using the `compareTo` method. (See the `BigDecimal` documentation for details.)

Writing a `compareTo` method is similar to writing an `equals` method, but there are a few key differences. Because the `Comparable` interface is parameterized, the `compareTo` method is statically typed, so you don't need to type check or cast its argument. If the argument is of the wrong type, the invocation won't even compile. If the argument is `null`, the invocation should throw a `NullPointerException`, and it will, as soon as the method attempts to access its members.

In a `compareTo` method, fields are compared for order rather than equality. To compare object reference fields, invoke the `compareTo` method recursively. If a field does not implement `Comparable` or you need a nonstandard ordering, use a `Comparator` instead. You can write your own comparator or use an existing one, as in this `compareTo` method for `CaseInsensitiveString` in Item 10:

```
// Single-field Comparable with object reference field
public final class CaseInsensitiveString
        implements Comparable<CaseInsensitiveString> {
    public int compareTo(CaseInsensitiveString cis) {
        return String.CASE_INSENSITIVE_ORDER.compare(s, cis.s);
    }
    ... // Remainder omitted
}
```

Note that `CaseInsensitiveString` implements `Comparable<CaseInsensitiveString>`. This means that a `CaseInsensitiveString` reference can be compared only to another `CaseInsensitiveString` reference. This is the normal pattern to follow when declaring a class to implement `Comparable`.

Prior editions of this book recommended that `compareTo` methods compare integral primitive fields using the relational operators `<` and `>`, and floating point primitive fields using the static methods `Double.compare` and `Float.compare`. In Java 7, static `compare` methods were added to all of Java's boxed primitive classes. **Use of the relational operators `<` and `>` in `compareTo` methods is verbose and error-prone and no longer recommended.**

If a class has multiple significant fields, the order in which you compare them is critical. Start with the most significant field and work your way down. If a comparison results in anything other than zero (which represents equality), you're done; just return the result. If the most significant field is equal, compare the next-most-significant field, and so on, until you find an unequal field or compare the least significant field. Here is a `compareTo` method for the `PhoneNumber` class in Item 11 demonstrating this technique:

```
// Multiple-field Comparable with primitive fields
public int compareTo(PhoneNumber pn) {
    int result = Short.compare(areaCode, pn.areaCode);
    if (result == 0)  {
        result = Short.compare(prefix, pn.prefix);
        if (result == 0)
            result = Short.compare(lineNum, pn.lineNum);
    }
    return result;
}
```

In Java 8, the `Comparator` interface was outfitted with a set of *comparator construction methods*, which enable fluent construction of comparators. These comparators can then be used to implement a `compareTo` method, as required by the `Comparable` interface. Many programmers prefer the conciseness of this approach, though it does come at a modest performance cost: sorting arrays of `PhoneNumber` instances is about 10% slower on my machine. When using this approach, consider using Java's *static import* facility so you can refer to static comparator construction methods by their simple names for clarity and brevity. Here's how the `compareTo` method for `PhoneNumber` looks using this approach:

```
// Comparable with comparator construction methods
private static final Comparator<PhoneNumber> COMPARATOR =
        comparingInt((PhoneNumber pn) -> pn.areaCode)
          .thenComparingInt(pn -> pn.prefix)
          .thenComparingInt(pn -> pn.lineNum);

public int compareTo(PhoneNumber pn) {
    return COMPARATOR.compare(this, pn);
}
```

This implementation builds a comparator at class initialization time, using two comparator construction methods. The first is `comparingInt`. It is a static method that takes a *key extractor function* that maps an object reference to a key of type `int` and returns a comparator that orders instances according to that key. In the previous example, `comparingInt` takes a *lambda* () that extracts the area code from a `PhoneNumber` and returns a `Comparator<PhoneNumber>` that orders phone numbers according to their area codes. Note that the lambda explicitly specifies the type of its input parameter (`PhoneNumber pn`). It turns out that in this situation, Java's type inference isn't powerful enough to figure the type out for itself, so we're forced to help it in order to make the program compile.

If two phone numbers have the same area code, we need to further refine the comparison, and that's exactly what the second comparator construction method, `thenComparingInt`, does. It is an instance method on `Comparator` that takes an `int` key extractor function, and returns a comparator that first applies the original comparator and then uses the extracted key to break ties. You can stack up as many calls to `thenComparingInt` as you like, resulting in a *lexicographic ordering*. In the example above, we stack up two calls to `thenComparingInt`, resulting in an ordering whose secondary key is the prefix and whose tertiary key is the line number. Note that we did *not* have to specify the parameter type of the key extractor function passed to either of the calls to `thenComparingInt`: Java's type inference was smart enough to figure this one out for itself.

The `Comparator` class has a full complement of construction methods. There are analogues to `comparingInt` and `thenComparingInt` for the primitive types `long` and `double`. The `int` versions can also be used for narrower integral types, such as `short`, as in our `PhoneNumber` example. The `double` versions can also be used for `float`. This provides coverage of all of Java's numerical primitive types.

There are also comparator construction methods for object reference types. The static method, named `comparing`, has two overloadings. One takes a key extractor and uses the keys' natural order. The second takes both a key extractor and a comparator to be used on the extracted keys. There are three overloadings of the instance method, which is named `thenComparing`. One overloading takes only a comparator and uses it to provide a secondary order. A second overloading takes only a key extractor and uses the key's natural order as a secondary order. The final overloading takes both a key extractor and a comparator to be used on the extracted keys.

Occasionally you may see `compareTo` or `compare` methods that rely on the fact that the difference between two values is negative if the first value is less than the second, zero if the two values are equal, and positive if the first value is greater. Here is an example:

```
// BROKEN difference-based comparator - violates transitivity!
static Comparator<Object> hashCodeOrder = new Comparator<>() {
    public int compare(Object o1, Object o2) {
        return o1.hashCode() - o2.hashCode();
    }
};
```

Do not use this technique. It is fraught with danger from integer overflow and IEEE 754 floating point arithmetic artifacts [JLS 15.20.1, 15.21.1]. Furthermore, the resulting methods are unlikely to be significantly faster than those written using the techniques described in this item. Use either a static `compare` method:

```
// Comparator based on static compare method
static Comparator<Object> hashCodeOrder = new Comparator<>() {
    public int compare(Object o1, Object o2) {
        return Integer.compare(o1.hashCode(), o2.hashCode());
    }
};
```

or a comparator construction method:

```
// Comparator based on Comparator construction method
static Comparator<Object> hashCodeOrder =
        Comparator.comparingInt(o -> o.hashCode());
```

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.