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 these contracts 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 6. While not an `Object` method, `Comparable.compareTo` is discussed in this chapter because it has a similar character.

Item 7: Obey the general contract when overriding `equals`

Overriding the `equals` method seems simple, but there are many ways to get it wrong, and the consequences can be dire. The easiest way to avoid problems is not to override the `equals` method, in which case each instance 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 that represent active entities rather than values, such as `Thread`. The `equals` implementation provided by `Object` has exactly the right behavior for these classes.
- **You don't care whether the class provides a "logical equality" test.** For example, `java.util.Random` could have overridden `equals` to check whether two `Random` instances would produce the same sequence of random numbers going forward, but the designers didn't think that clients would need or want this functionality. Under these circumstances, the `equals` implementation inherited from `Object` is adequate.

- **A superclass has already overridden `equals`, and the behavior inherited from the superclass 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.** Arguably, the `equals` method *should* be overridden under these circumstances, in case it is accidentally invoked someday:

```
public boolean equals(Object o) {
    throw new UnsupportedOperationException();
}
```

So when is it appropriate to override `Object.equals`? When a class has a notion of *logical equality* that differs from mere object identity, and a superclass has not already overridden `equals` to implement the desired behavior. This is generally the case for *value classes*, such as `Integer` or `Date`. 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 of the class 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 the *typesafe enum* (Item 21). Because typesafe enum classes guarantee that at most one object exists with each value, `Object`'s `equals` method is equivalent to a logical `equals` method for such classes.

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

> The equals method implements an *equivalence relation*:
>
> - It is *reflexive*: For any reference value `x`, `x.equals(x)` must return `true`.
> - It is *symmetric*: For any reference values `x` and `y`, `x.equals(y)` must return `true` if and only if `y.equals(x)` returns `true`.
> - It is *transitive*: For any reference values `x`, `y`, and `z`, if `x.equals(y)` returns `true` and `y.equals(z)` returns `true`, then `x.equals(z)` must return `true`.
> - It is *consistent*: For any reference values `x` and `y`, multiple invocations of `x.equals(y)` consistently return `true` or consistently return `false`, provided no information used in `equals` comparisons on the object 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 evils of violating the `equals` contract, let's go over the contract in detail. The good news is that, appearances notwithstanding, the contract really isn't very complicated. Once you understand it, it's not hard to adhere to it. Let's examine the five requirements in turn:

Reflexivity—The first requirement says merely that an object must be equal to itself. It is hard to imagine violating this requirement unintentionally. If you were to violate it and then add an instance of your class to a collection, the collection's `contains` method would almost certainly say that the collection did not 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:

```
/**
 * Case-insensitive string. Case of the original string is
 * preserved by toString, but ignored in comparisons.
 */
public final class CaseInsensitiveString {
    private String s;

    public CaseInsensitiveString(String s) {
        if (s == null)
            throw new NullPointerException();
        this.s = s;
    }

    // Broken - violates symmetry!
    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-sensitive 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 list = new ArrayList();
list.add(cis);
```

What does `list.contains(s)` return at this point? Who knows? In Sun's current 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 run-time 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 to give it a single return:

```
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 programmer who creates a subclass that adds a new *aspect* 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 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;
    }

    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 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!
public boolean equals(Object o) {
    if (!(o instanceof ColorPoint))
       return false;
    ColorPoint cp = (ColorPoint)o;
    return super.equals(o) && cp.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.
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
    ColorPoint cp = (ColorPoint)o;
    return super.equals(o) && cp.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);
```

At this point, `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.

So what's the solution? It turns out that this is a fundamental problem of equivalence relations in object-oriented languages. **There is simply no way to extend an instantiable class and add an aspect while preserving the `equals` contract.** There is, however, a fine workaround. Follow the advice of Item 14, "Favor composition over inheritance." Instead of having `ColorPoint` extend

Point, give ColorPoint a private Point field and a public *view* method (Item 4) that returns the point at the same position as this color point:

```
// Adds an aspect without violating the equals contract
public class ColorPoint {
   private Point point;
   private Color color;

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

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

   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 subclass an instantiable class and add an aspect. For example, java.sql.Timestamp subclasses java.util.Date adding 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 the programmer against mixing dates and timestamps. While you won't get into trouble as long as you don't mix them, there's nothing preventing you from doing so, and the resulting errors could be hard to debug. The TimeStamp class is an anomaly and should not be emulated.

Note that you *can* add an aspect 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 20, "Replace unions with class hierarchies." For example, you could have an abstract Shape class with no aspects, a Circle subclass that adds a radius field, and a Rectangle subclass that adds

`length` and `width` fields. Problems of the sort just shown will not occur as long as it is impossible to create an instance of the superclass.

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. This isn't so much a true requirement as a reminder that 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 13). 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.

"Non-nullity"—The final requirement, which in the absence of a name I have taken the liberty of calling "non-nullity," 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 does not allow this. Many classes have `equals` methods that guard against it with an explicit test for `null`:

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

This test is not necessary. To test its argument for equality, the `equals` method must first cast the argument to an appropriate type so its accessors may 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:

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

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, 15.19.2]. Therefore the type check will return `false` if `null` is passed in, so you don't need a separate `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 is of 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. The collection interfaces `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 to see 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 significant 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, invoke the `equals` method recursively; for `float` fields, translate to `int` values using `Float.floatToIntBits` and compare the `int` values using the == operator; for `double` fields, translate to `long` values using `Double.doubleToLongBits` and compare the `long` values using the == operator. (The special treatment of `float` and `double` fields is made necessary by the existence of `Float.NaN`, `-0.0f`, and the analogous `double` constants; see the `Float.equals` documentation for details.) For array fields, apply these guidelines to each element. Some object reference fields may legitimately contain `null`. To avoid the possibility of a `NullPointerException`, use the following idiom to compare such fields:

```
(field == null ? o.field == null : field.equals(o.field))
```

This alternative may be faster if `field` and `o.field` are often identical object references:

```
(field == o.field || (field != null && field.equals(o.field)))
```

For some classes, like `CaseInsensitiveString` shown earlier, the field comparisons are more complex than simple equality tests. It should be apparent from the specification for a class if this is the case. If so, you may want to store

a *canonical form* in each object, so that the `equals` method can do cheap exact comparisons on these canonical forms rather than more costly inexact comparisons. This technique is most appropriate for *immutable* classes (Item 13), as the canonical form would have to be kept up to date if the object could change.

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 `Object` fields used to synchronize operations. You need not compare *redundant fields*, which can be calculated from "significant fields," but doing so may improve the performance of the `equals` method. If a redundant 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.

5. **When you are finished writing your equals method, ask yourself three questions: Is it symmetric, is it transitive, and is it consistent?** (The other two properties generally take care of themselves.) If not, figure out why these properties fail to hold, and modify the method accordingly.

For a concrete example of an `equals` method constructed according to the above recipe, see `PhoneNumber.equals` in Item 8. Here are a few final caveats:

- **Always override `hashCode` when you override `equals`** (Item 8).
- **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 write an `equals` method that relies on unreliable resources.** It's extremely difficult to satisfy the consistency requirement if you do this. For example, `java.net.URL`'s `equals` method relies on the IP addresses of the hosts in URLs being compared. 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 it has caused problems in practice. (Unfortunately, this behavior cannot be changed due to compatibility requirements.) With few exceptions, `equals` methods should perform deterministic computations on memory-resident objects.

- **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 the following, and then spend hours puzzling over why it doesn't work properly:

```
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 26). It is acceptable to provide such a "strongly typed" `equals` method *in addition* to the normal one as long as the two methods return the same result but there is no compelling reason to do so. It may provide minor performance gains under certain circumstances, but it isn't worth the added complexity (Item 37).

Item 8: Always override hashCode when you override equals

A common source of bugs is the failure to override the `hashCode` method. **You must override `hashCode` in every class that overrides `equals`.** Failure to do so will result in a violation of the general contract for `Object.hashCode`, which will prevent your class from functioning properly in conjunction with all hash-based collections, including `HashMap`, `HashSet`, and `Hashtable`.

Here is the contract, copied from the `java.lang.Object` specification:

- Whenever it is invoked on the same object more than once during an execution of an application, the `hashCode` method must consistently return the same integer, provided no information used in `equals` comparisons on the object is modified. This integer need not remain consistent from one execution of an application to another execution of the same application.
- If two objects are equal according to the `equals(Object)` method, then calling the `hashCode` method on each of the two objects must produce the same integer result.
- It is not required that if two objects are unequal according to the `equals(Object)` method, then calling the `hashCode` method on each of the two objects must produce distinct integer results. However, the programmer should be aware that producing distinct integer 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 the class's `equals` method, but to the `Object` class'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, consider the following simplistic `PhoneNumber` class, whose `equals` method is constructed according to the recipe in Item 7:

```
public final class PhoneNumber {
    private final short areaCode;
    private final short exchange;
    private final short extension;

    public PhoneNumber(int areaCode, int exchange,
                       int extension) {
        rangeCheck(areaCode,   999, "area code");
        rangeCheck(exchange,   999, "exchange");
        rangeCheck(extension, 9999, "extension");
```

```
        this.areaCode  = (short) areaCode;
        this.exchange  = (short) exchange;
        this.extension = (short) extension;
    }

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

    public boolean equals(Object o) {
        if (o == this)
            return true;
        if (!(o instanceof PhoneNumber))
            return false;
        PhoneNumber pn = (PhoneNumber)o;
        return pn.extension == extension &&
               pn.exchange  == exchange  &&
               pn.areaCode  == areaCode;
    }

    // No hashCode method!

    ... // Remainder omitted
}
```

Suppose you attempt to use this class with a `HashMap`:

```
Map m = new HashMap();
m.put(new PhoneNumber(408, 867, 5309), "Jenny");
```

At this point, you might expect `m.get(new PhoneNumber(408, 867, 5309))` to return `"Jenny"`, but 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 looks for the phone number in a different hash bucket from the one in which it was stored by the `put` method. Fixing this problem is as simple as providing a proper `hashCode` method for the `PhoneNumber` class.

So what should a `hashCode` method look like? It's trivial to write one that is legal but not good. This one, for example, is always legal, but it should never be used:

```
// The worst possible legal hash function - never use!
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 run instead 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 objects. This is exactly what is meant by the third provision of the hashCode contract. Ideally, a hash function should distribute any reasonable collection of unequal instances uniformly across all possible hash values. Achieving this ideal can be extremely difficult. Luckily it is not too difficult to achieve a fair approximation. Here is a simple recipe:

1. Store some constant nonzero value, say 17, in an `int` variable called `result`.
2. For each significant field `f` in your object (each field taken into account by the `equals` method, that is), do the following:
   a. Compute an `int` hash code `c` for the field:
      i. If the field is a `boolean`, compute `(f ? 0 : 1)`.
      ii. If the field is a `byte`, `char`, `short`, or `int`, compute `(int)f`.
      iii. If the field is a `long`, compute `(int)(f ^ (f >>> 32))`.
      iv. If the field is a `float` compute `Float.floatToIntBits(f)`.
      v. If the field is a `double`, compute `Double.doubleToLongBits(f)`, and then hash the resulting `long` as in step 2.a.iii.
      vi. 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`, return `0` (or some other constant, but 0 is traditional).

vii. If the field is an array, treat it as if each element were a separate field. That is, compute a hash code for each significant element by applying these rules recursively, and combine these values as described in step 2.b.

b. Combine the hash code `c` computed in step a into `result` as follows:

```
result = 37*result + c;
```

3. Return `result`.
4. When you are done writing the `hashCode` method, ask yourself whether equal instances have equal hash codes. If not, figure out why and fix the problem.

It is acceptable to exclude *redundant fields* from the hash code computation. In other words, it is acceptable to exclude any field whose value can be computed from fields that are included in the computation. It is *required* that you exclude any fields that are not used in equality comparisons. Failure to exclude these fields may result in a violation of the second provision of the `hashCode` contract.

A nonzero initial value is used in step 1, so the hash value will be affected by initial fields whose hash value, as computed in step 2.a, is zero. If zero was used as the initial value in step 1, the overall hash value would be unaffected by any such initial fields, which could increase collisions. The value 17 is arbitrary.

The multiplication in step 2.b makes the hash value depend on the order of the fields, which results in a much better hash function if the class contains multiple similar fields. For example, if the multiplication were omitted from a `String` hash function built according to this recipe, all anagrams would have identical hash codes. The multiplier 37 was chosen because it is an odd prime. If it was even and the multiplication overflowed, information would be lost because multiplication by two is equivalent to shifting. The advantages of using a prime number are less clear, but it is traditional to use primes for this purpose.

Let's apply this recipe to the `PhoneNumber` class. There are three significant fields, all of type `short`. A straightforward application of the recipe yields this hash function:

```
public int hashCode() {
    int result = 17;
    result = 37*result + areaCode;
    result = 37*result + exchange;
    result = 37*result + extension;
    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 should be clear that equal `PhoneNumber` instances have equal hash codes. This method is, in fact, a perfectly reasonable `hashCode` implementation for `PhoneNumber`, on a par with those in the Java platform libraries as of release 1.4. It is simple, is reasonably fast, and does a reasonable job of dispersing unequal phone numbers into different hash buckets.

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* it the first time `hashCode` is invoked (Item 48). It is not clear that our `PhoneNumber` class merits this treatment, but just to show you how it's done:

```
// Lazily initialized, cached hashCode
private volatile int hashCode = 0;  // (See Item 48)

public int hashCode() {
    if (hashCode == 0) {
        int result = 17;
        result = 37*result + areaCode;
        result = 37*result + exchange;
        result = 37*result + extension;
        hashCode = result;
    }
    return hashCode;
}
```

While the recipe in this item yields reasonably good hash functions, it does not yield state-of-the-art hash functions, nor do the Java platform libraries provide such hash functions as of release 1.4. Writing such hash functions is a topic of active research and an activity best left to mathematicians and theoretical computer scientists. Perhaps a later release of the Java platform will provide state-of-the-art hash functions for its classes and utility methods to allow average programmers to construct such hash functions. In the meantime, the techniques described in this item should be adequate for most applications.

Do not be tempted to exclude significant parts of an object from the hash code computation to improve performance. While the resulting hash function may run faster, its quality may degrade to the point where hash tables become unusably slow. In particular, the hash function may, in practice, be confronted

with a large collection of instances that differ largely in the regions that you've chosen to ignore. If this happens, the hash function will map all of the instances to a very few hash codes, and hash-based collections will display quadratic performance. This is not just a theoretical problem. The `String` hash function implemented in all Java platform releases prior to release 1.2 examined at most sixteen characters, evenly spaced throughout the string, starting with the first character. For large collections of hierarchical names such as URLs, this hash function displayed exactly the pathological behavior noted here.

Many classes in the Java platform libraries, such as `String`, `Integer`, and `Date`, specify the exact value returned by their `hashCode` method as a function of the instance value. This is generally *not* a good idea, as it severely limits your ability to improve the hash function in future releases. If you leave the details of a hash function unspecified and a flaw is found in it, you can fix the hash function in the next release without fear of breaking compatibility with clients who depend on the exact values returned by the hash function.

Item 9: Always override toString

While java.lang.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 "(408) 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 important as obeying the equals and hashCode contracts (Item 7, Item 8), **providing a good toString implementation makes your class much more pleasant to use**. The toString method is automatically invoked when your object is passed to println, the string concatenation operator (+), or, as of release 1.4, assert. If you've provided a good toString method, generating a useful diagnostic message is as easy as:

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

Programmers will generate diagnostic messages in this fashion whether or not you override toString, but the messages won't be intelligible 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=(408) 867-5309}"?

When practical, the toString method should return *all* of the interesting information contained in the object, as in the phone number example just shown. 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 white pages (1487536 listings)" or "Thread[main, 5,main]". Ideally, the string should be self-explanatory. (The Thread example flunks this test.)

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 numbers or matrices. 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 XML documents. If you specify the format, it's usually a good idea to provide a matching `String` constructor (or static factory, see Item 1), 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 primitive wrapper 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 failing 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 8:

```
/**
 * Returns the string representation of this phone number.
 * The string consists of fourteen characters whose format
 * is "(XXX) YYY-ZZZZ", where XXX is the area code, YYY is
 * the extension, and ZZZZ is the exchange.  (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 exchange is 123, the last
 * four characters of the string representation will be "0123".
 *
 * Note that there is a single space separating the closing
 * parenthesis after the area code from the first digit of the
 * exchange.
 */
public String toString() {
    return "(" + toPaddedString(areaCode, 3) + ") " +
            toPaddedString(exchange,  3) + "-" +
            toPaddedString(extension, 4);
}

private static String[] ZEROS =
    {"", "0", "00", "000", "0000", "00000",
    "000000", "0000000", "00000000", "000000000"};
```

```
/**
 * Translates an int to a string of the specified length,
 * padded with leading zeros.  Assumes i >= 0,
 * 1 <= length <= 10, and Integer.toString(i) <= length.
 */
private static String toPaddedString(int i, int length) {
    String s = Integer.toString(i);
    return ZEROS[length - s.length()] + s;
}
```

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]"
 */
public String toString() { ... }
```

After reading this comment, programmers who produce code or persistent data that depend 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, **it is always a good idea to provide programmatic access to all of the information contained in the value returned by `toString`.** For example, the `PhoneNumber` class should contain accessors for the area code, exchange, and extension. 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.

Item 10: Override `clone` judiciously

The `Cloneable` interface was intended as a *mixin interface* (Item 16) for objects 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 35), invoke the `clone` method on an object merely because it implements `Cloneable`. Even a reflective invocation may fail, as there is no guarantee that the object has an accessible `clone` method. Despite this flaw and others, the facility is in sufficiently wide use that 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 briefly discusses 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 the case of `Cloneable`, however, it modifies the behavior of a protected method on a superclass.

In order for implementing the `Cloneable` interface to have any effect on a class, it and all of its superclasses must obey a fairly complex, unenforceable, and largely undocumented protocol. The resulting mechanism is *extralinguistic*: It creates an object without calling a constructor.

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

> 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. Copying an object will typically entail creating a new instance of its class, but it may require copying of internal data structures as well. No constructors are called.

There are a number of problems with this contract. The provision that "no constructors are called" is too strong. A well-behaved `clone` method can call constructors to create objects internal to the `clone` under construction. If the class is final, `clone` can even return an object created by a constructor.

The provision that `x.clone().getClass()` should generally be identical to `x.getClass()`, however, is too weak. In practice, programmers assume that if they extend a class and invoke `super.clone` from the subclass, the returned object will be an instance of the subclass. The *only* way a superclass can provide this functionality is to return an object obtained by calling `super.clone`. If a `clone` method returns an object created by a constructor, it will have the wrong class. Therefore, **if you override the `clone` method in a nonfinal class, you should return an object obtained by invoking `super.clone`.** If all of a class's superclasses obey this rule, then invoking `super.clone` will eventually invoke `Object`'s `clone` method, creating an instance of the right class. This mechanism is vaguely similar to automatic constructor chaining, except that it isn't enforced.

The `Cloneable` interface does not, as of Release 1.3, spell out the responsibilities that a class takes on when it implements this interface. The specification says nothing beyond the manner in which implementing the interface affects the behavior of `Object`'s `clone` implementation. **In practice, a class that implements `Cloneable` is expected to provide a properly functioning public `clone` method.** It is not, in general, possible to do so unless all of the class's superclasses provide a well-behaved `clone` implementation, whether public or protected.

Suppose you want to implement `Cloneable` in a class whose superclasses provide well-behaved `clone` methods. The object you get from `super.clone()` may or may not be close to what you'll eventually return, depending on the nature of the class. This object will be, from the standpoint of each superclass, a fully functional clone of the original object The fields declared in your class (if any) will have values identical to those of the object being cloned. 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 8. In this case, all you need do is provide public access to `Object`'s protected `clone` method:

```
public Object clone() {
    try {
        return super.clone();
    } catch(CloneNotSupportedException e) {
        throw new Error("Assertion failure");  // Can't happen
    }
}
```

If, however, your object contains fields that refer to mutable objects, using this `clone` implementation can be disastrous. For example, consider the `Stack` class in Item 5:

```
public class Stack {
    private Object[] elements;
    private int size = 0;

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

    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) {
            Object oldElements[] = elements;
            elements = new Object[2 * elements.length + 1];
            System.arraycopy(oldElements, 0, elements, 0, size);
        }
    }
}
```

Suppose you want to make this class cloneable. If its `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 `ArrayIndexOutOfBoundsException`.

This situation could never occur as a result of calling the sole constructor in the `Stack` class. **In effect, the `clone` method functions as another 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 by calling `clone` recursively on the `elements` array:

```
public Object clone() throws CloneNotSupportedException {
    Stack result = (Stack) super.clone();
    result.elements = (Object[]) elements.clone();
    return result;
}
```

Note that this solution would not work if the `buckets` field were final because the `clone` method would be prohibited from assigning a new value to the field. This is a fundamental problem: **the `clone` 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 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 or is `null` if the bucket is empty. 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 {
        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 - results in shared internal state!
public Object clone() throws CloneNotSupportedException {
    HashTable result = (HashTable) super.clone();
    result.buckets = (Entry[]) buckets.clone();
    return result;
}
```

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 individually. Here is one common approach:

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

    private static class Entry {
        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());
        }
    }

    public Object clone() throws CloneNotSupportedException {
        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] = (Entry)
                    buckets[i].deepCopy();

        return result;
    }
    ... // 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 deep-copy 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 objects is to call `super.clone`, set all of the fields in the resulting object to their virgin state, and then call higher-level methods to regenerate the state of the 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 doesn't run quite as fast as one that directly manipulates the innards of the object and its clone.

Like a constructor, a `clone` method should not invoke any nonfinal methods on the clone under construction (Item 15). If `clone` invokes an overridden method, this method will execute before the subclass in which it is defined 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 `clone` methods may omit this declaration. The `clone` methods of final classes should omit the declaration because methods that don't throw checked exceptions are more pleasant to use than those that do (Item 41). If an extendable class, especially one designed for inheritance (Item 15), overrides the

`clone` method, the overriding `clone` method should include the declaration to throw `CloneNotSupportedException`. Doing this allows subclasses to opt out of clonability gracefully, by providing the following `clone` method:

```
// Clone method to guarantee that instances cannot be cloned
public final Object clone() throws CloneNotSupportedException {
    throw new CloneNotSupportedException();
}
```

It is not essential that the foregoing advice be followed, as the `clone` method of a subclass that doesn't want to be cloned can always throw an unchecked exception, such as `UnsupportedOperationException`, if the `clone` method it overrides is not declared to throw `CloneNotSupportedException`. Common practice, however, dictates that `CloneNotSupportedException` is the correct exception to throw under these circumstances.

To recap, all classes that implement `Cloneable` should override `clone` with a public method. This public method should first call `super.clone` and then fix any fields that need fixing. Typically, this means copying any mutable objects that comprise the internal "deep structure" of the object being cloned and replacing the references to these objects with references to the copies. While these internal copies can generally 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 probably 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 or a field representing the object's creation time 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 implements `Cloneable`, you have little choice but to implement a well-behaved `clone` method. Otherwise, **you are probably better off providing some alternative means of object copying or simply not providing the capability**. For example, it doesn't make much sense for immutable classes to support object copying, because copies would be virtually indistinguishable from the original.

A fine approach to object copying is to provide a *copy constructor.* A copy constructor is simply a constructor that takes a single argument whose type is the class containing the constructor, for example,

```
public Yum(Yum yum);
```

A minor variant is to provide a static factory in place of a constructor:

```
public static Yum newInstance(Yum yum);
```

The copy constructor approach and its static factory variant have many advantages over `Cloneable/clone`: They do not rely on a risk-prone extralinguistic object creation mechanism; they do not demand unenforceable adherence to ill-documented conventions; they do not conflict with the proper use of final fields; they do not require the client to catch an unnecessary checked exception; and they provide a statically typed object to the client. While it is impossible to put a copy constructor or static factory in an interface, `Cloneable` fails to function as an interface because it lacks a public `clone` method. Therefore you aren't giving up interface functionality by using a copy constructor instead of a `clone` method.

Furthermore, a copy constructor (or static factory) can take an argument whose type is an appropriate interface implemented by the class. For example, all general-purpose collection implementations, by convention, provide a copy constructor whose argument is of type `Collection` or `Map`. Interface-based copy constructors allow the client to choose the implementation of the copy, rather than forcing the client to accept the implementation of the original. For example, suppose you have a `LinkedList l`, and you want to copy it as an `ArrayList`. The `clone` method does not offer this functionality, but it's easy with a copy constructor: `new ArrayList(l)`.

Given all of the problems associated with `Cloneable`, it is safe to say that other interfaces should not extend it and that classes designed for inheritance (Item 15) should not implement it. Because of its many shortcomings, some expert programmers simply choose never to override the `clone` method and never to invoke it except, perhaps, to copy arrays cheaply. Be aware that if you do not at least provide a well-behaved *protected* `clone` method on a class designed for inheritance, it will be impossible for subclasses to implement `Cloneable`.

Item 11: 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 `java.lang.Comparable` interface. It is similar in character to `Object`'s `equals` method, except that it permits order comparisons in addition to simple equality comparisons. 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 s = new TreeSet();
        s.addAll(Arrays.asList(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 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 strongly consider implementing this interface. This item tells you how to go about it.

The general contract for the `compareTo` method is similar in character to that of the `equals` method. Here it is, copied from the specification for `Comparable`:

> 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 `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 strictly 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."

Do not be put off by the mathematical nature of this contract. Like the `equals` contract (Item 7), the `compareTo` contract isn't as complicated as it looks. Within a class, any reasonable ordering relation will satisfy the `compareTo` contract. Across classes, `compareTo`, unlike `equals`, doesn't have to work: It is permitted to throw `ClassCastException` if the two object references being compared refer to objects of different classes. Usually, that is exactly what `compareTo` should do under these circumstances. While the contract does not preclude interclass comparisons, there are, as of release 1.4, no classes in the Java platform libraries that support them.

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, transitivity, and non-nullity. Therefore the same caveat applies: There is simply no way to extend an instantiable class with a new aspect while preserving the `compareTo` contract (Item 7). The same workaround applies too. If you want to add a significant aspect to a class that implements `Comparable`, don't extend it; write an unrelated class that contains a field of the first class. Then provide a "view" method that returns this field. This frees you to implement whatever `compareTo` method you like on the second class, while allowing its client to view an instance of the second class as an instance of the first class when needed.

The final paragraph of the `compareTo` contract, which is a strong suggestion rather than a true provision, 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 `Float` class, whose `compareTo` method is inconsistent with equals. If you create a `HashSet` and add `new Float(-0.0f)` and `new Float(0.0f)`, the set will contain two elements because the two `Float` 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 `Float` instances are equal when compared using the `compareTo` method. (See the `Float` documentation for details.)

Writing a `compareTo` method is similar to writing an `equals` method, but there are a few key differences. You don't need to type check the argument prior to casting. If the argument is not of the appropriate type, the `compareTo` method *should* throw a `ClassCastException`. If the argument is `null`, the `compareTo` method *should* throw a `NullPointerException`. This is precisely the behavior that you get if you just cast the argument to the correct type and then attempt to access its members.

The field comparisons themselves are order comparisons rather than equality comparisons. Compare object reference fields by invoking the `compareTo` method recursively. If a field does not implement `Comparable` or you need to use a nonstandard ordering, you can use an explicit `Comparator` instead. Either write your own or use a preexisting one as in this `compareTo` method for the `CaseInsensitiveString` class in Item 7:

```
public int compareTo(Object o) {
    CaseInsensitiveString cis = (CaseInsensitiveString)o;
    return String.CASE_INSENSITIVE_ORDER.compare(s, cis.s);
}
```

Compare primitive fields using the relational operators < and >, and arrays by applying these guidelines to each element. If a class has multiple significant fields, the order in which you compare them is critical. You must 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 fields are equal, go on to compare the next-most-significant fields, and so on. If all fields are equal, the objects are equal; return zero. The technique is demonstrated by this `compareTo` method for the `PhoneNumber` class in Item 8:

```
public int compareTo(Object o) {
    PhoneNumber pn = (PhoneNumber)o;

    // Compare area codes
    if (areaCode < pn.areaCode)
        return -1;
    if (areaCode > pn.areaCode)
        return  1;

    // Area codes are equal, compare exchanges
    if (exchange < pn.exchange)
        return -1;
    if (exchange > pn.exchange)
        return  1;

    // Area codes and exchanges are equal, compare extensions
    if (extension < pn.extension)
        return -1;
    if (extension > pn.extension)
        return  1;

    return 0;  // All fields are equal
}
```

While this method works fine, it can be improved. Recall that the contract for `compareTo` does not specify the magnitude of the return value, only the sign. You can take advantage of this to simplify the code and probably make it run a bit faster:

```
public int compareTo(Object o) {
    PhoneNumber pn = (PhoneNumber)o;

    // Compare area codes
    int areaCodeDiff = areaCode - pn.areaCode;
    if (areaCodeDiff != 0)
        return areaCodeDiff;

    // Area codes are equal, compare exchanges
    int exchangeDiff = exchange - pn.exchange;
    if (exchangeDiff != 0)
        return exchangeDiff;

    // Area codes and exchanges are equal, compare extensions
    return extension - pn.extension;
}
```

This trick works fine here but should be used with extreme caution. Don't do it unless you're certain that the field in question cannot be negative or, more generally, that the difference between the lowest and highest possible field values is less than or equal to `INTEGER.MAX_VALUE` (2^{31}-1). The reason this trick does not work in general is that a signed 32-bit integer is not big enough to represent the difference between two arbitrary signed 32-bit integers. If `i` is a large positive `int` and `j` is a large negative `int`, `(i-j)` will overflow and return a negative value. The resulting `compareTo` method will not work. It will return nonsensical results for some arguments, and it will violate the first and second provisions of the `compareTo` contract. This is not a purely theoretical problem; it has caused failures in real systems. These failures can be difficult to debug, as the broken `compareTo` method works properly for many input values.