Object Comparison 14

Chapter Topics

- Overview of selected convenience methods of the `Objects` class
- How to correctly override and implement the contract of the `equals()` method of the `Object` class
- How to correctly override and implement the contract of the `hash-Code()` method of the `Object` class
- Comparing objects whose natural order is defined by the contract of the `com-pareTo()` method of the `java.lang.Comparable<T>` interface
- Comparing objects using a comparator that defines a total order according to the contract of the `compare()` method of the `java.util.Comparator<T>` interface

Java SE 17 Developer Exam Objectives

[3.5] Implement inheritance, including abstract and sealed classes. Override methods, including that of Object class. Implement polymorphism and differentiate object type versus reference type. Perform type casting, identify object types using instanceof operator and pattern matching

§14.1, p. 743 §14.2, p. 744 §14.3, p. 753

- *Overriding methods of the* `Object` *class is covered in this chapter.*
- *For inheritance, abstract and sealed classes, overriding methods, polymorphism and type casting, static and dynamic type of a reference,* `instanceof` *type comparison and pattern match operators, see* Chapter 5, p. 189.

[5.1] Create Java arrays, List, Set, Map and Deque collections, and add, remove, update, retrieve and sort their elements

§14.4, p. 761 §14.5, p. 769

- `Comparable` *and* `Comparator` *interfaces for comparing objects are covered in this chapter.*
- *For arrays, see* §3.9, p. 117.
- *For* `ArrayList`, *see* Chapter 12, p. 643.

- *For lists, sets, maps, deques, and sorted collections and arrays, see*

Chapter 15, p. 781.

Java SE 11 Developer Exam Objectives	
[5.3] Sort collections and arrays using Comparator and Comparable interfaces - `Comparable` *and* `Comparator` *interfaces are covered in this chapter.* - *For sorted collections and arrays, see* **Chapter 15, p. 781**.	§14.4, p. 761 §14.5, p. 769

This chapter covers the important topic of comparing objects. Many useful operations require that the objects can be compared for *object value equality* and *ranked* according to some meaningful criteria. Typical examples are searching and sorting algorithms for objects maintained in arrays, collections, and maps. Overriding the methods `equals()` and `hashCode()` in the `Object` class and the methods `compareTo()` and `compare()` in the interfaces `Comparable<T>` and `Comparator<T>`, respectively, is essential in this regard.

The `equals()` and `hashCode()` methods of the `Object` class provide specific contracts for objects, which the classes overriding the methods should honor. It is important to understand how and why a class should override the `equals()` and `hashCode()` methods.

The `compareTo()` method of the `Comparable<E>` interface defines the contract for comparing objects which classes can implement. In addition, the `compare()` method of the `Comparator<E>` interface allows other criteria to be defined for comparing objects.

Objects of a class that override the `equals()` method make it possible to search for such objects in arrays and lists. If they override the `hashCode()` method, they can also be searched for as elements in a set and as keys in a map. Implementing the `Comparable<E>` interface allows them to be sorted and maintained as elements in sorted collections and as keys in sorted maps. Collections and maps are covered in detail in **Chapter 15, p. 781**.

14.1 The `Objects` Class

The following `static` methods of the `java.util.Objects` class are convenient to use when comparing objects and computing the hash code for an object. They take away

the drudgery of checking references for `null` values before calling them and never throw a `NullPointerException`—that is, they are `null`-safe.

Click here to view code image

```
static boolean equals(Object obj1, Object obj2)
```

Returns `true` if the arguments are equal to each other; otherwise, it returns `false`. This means that if both arguments are `null`, `true` is returned and if only one argument is `null`, `false` is returned. Equality is determined by invoking the `equals()` method on the first argument: `obj1.equals(obj2)`. It is a convenience method for the `Object.equals()` method (**Example 14.3, p. 748**).

Click here to view code image

```
static int hash(Object... values)
static int hashCode(Object obj)
```

The first method generates a hash code for a sequence of specified values. It is useful for computing the hash code for objects containing multiple fields that are passed as arguments (**Example 14.6, p. 771**).

The second method computes and returns the hash code of a non-`null` argument; otherwise, it returns `0` for a `null` argument.

Both methods are convenience methods for the `Object.hashCode()` method.

Click here to view code image

```
static <T> int compare(T t1, T t2, Comparator<? super T> cmp)
```

Returns `0` if the arguments are identical; otherwise, it calls `cmp.compare(t1, t2)`, which is the abstract method in the `Comparator<T>` interface. It also returns `0` if both arguments are `null` (**Example 14.8, p. 777**).

14.2 Implementing the `equals()` Method

If every object is to be considered unique, then it is not necessary to override the `equals()` method in the `Object` class. This method implements *object reference*

equality. It implements the most discriminating equivalence relation possible on objects. Each instance of the class is only equal to itself.

Click here to view code image

```
public boolean equals(Object obj)
```

The `equals()` method in the `Object` class tests for *object reference equality*, the same as the `==` operator. It returns `true` only if the two references compared denote the same object—that is, if they are aliases. The `equals()` method is usually overridden to provide the semantics of *object value equality*, as in the case of the wrapper classes and the `String` class.

As a running example, we will implement different versions of a class for *version numbers*. A version number (VNO) for a software product comprises three pieces of information:

- A release number
- A revision number
- A patch number

The idea is that releases do not happen very often. Revisions take place more frequently than releases, but less frequently than code patches are issued. We can say that the release number is most *significant*. The revision number is less significant than the release number, and the patch number is the least significant of the three fields. This ranking would also be employed when ordering version numbers chronologically.

Example 14.1 *Not Overriding the* `Object.equals()` *Method*

Click here to view code image

```
/** A simple version number class */
public class SimpleVNO {

  private int release;
  private int revision;
  private int patch;

  public SimpleVNO(int release, int revision, int patch) {
    this.release  = release;
```

```
    this.revision = revision;
    this.patch    = patch;
  }

  public int getRelease()  { return this.release;  }
  public int getRevision() { return this.revision; }
  public int getPatch()    { return this.patch;    }

  @Override public String toString() {
    return "(" + release + "." + revision + "." + patch + ")";
  }
}
```

The class `SimpleVNO` in **Example 14.1** does not override the `equals()` method in the `Object` class. It only overrides the `toString()` method to generate a meaningful text representation for a version number.

The class `TestSimpleVNO` in **Example 14.2** creates objects of the class `SimpleVNO` in **Example 14.1** to test for *object reference equality* and *object value equality*. The output from **Example 14.2** demonstrates that all `SimpleVNO` objects are unique because the class `SimpleVNO` does not override the `equals()` method to provide any other equivalence relation. The result of the == operator is always `false` for object reference equality, since all `SimpleVNO` objects are unique. An attempt is made to compare `SimpleVNO` objects for object value equality using the `Object.equals()` method, but the implementation of the `Object.equals()` method tests for object reference equality, the same as the == operator. Not surprisingly, the `equals()` method returns `false` even for `SimpleVNO` objects that have the same state (`svno1` and `svno2`).

Example 14.2 *Implications of Not Overriding the* `Object.equals()` *Method*

Click here to view code image

```
import static java.lang.System.out;

public class TestSimpleVNO {
  public static void main(String[] args) {
    // Print name of version number class:
    out.println(SimpleVNO.class);

    // Three individual version numbers.
    SimpleVNO svno1 = new SimpleVNO(9,1,1);                    // (1)
    SimpleVNO svno2 = new SimpleVNO(9,1,1);                    // (2)
    SimpleVNO svno3 = new SimpleVNO(6,6,6);                    // (3)

    out.printf ("  svno1: %s, svno2: %s, svno3: %s%n", svno1, svno2, svno3);
```

```
    out.println("Test object reference equality:");                // (4)
    out.println("  svno1 == svno2:      " + (svno1 == svno2));     // (5)
    out.println("  svno1 == svno3:      " + (svno1 == svno3));     // (6)
    out.println("Test object value equality:");
    out.println("  svno1.equals(svno2): " + svno1.equals(svno2));  // (7)
    out.println("  svno1.equals(svno3): " + svno1.equals(svno3));  // (8)
  }
}
```

Output from the program:

Click here to view code image

```
class SimpleVNO
  svno1: (9.1.1), svno2: (9.1.1), svno3: (6.6.6)
Test object reference equality:
  svno1 == svno2:      false
  svno1 == svno3:      false
Test object value equality:
  svno1.equals(svno2): false
  svno1.equals(svno3): false
```

Equivalence Relation of the `equals()` Method

An implementation of the `equals()` method must satisfy the properties of an *equivalence relation*:

- *Reflexive*: For any reference `self`, `self.equals(self)` is always `true`.
- *Symmetric*: For any references `x` and `y`, if `x.equals(y)` is `true`, then `y.equals(x)` is `true`.
- *Transitive*: For any references `x`, `y`, and `z`, if both `x.equals(y)` and `y.equals(z)` are `true`, then `x.equals(z)` is `true`.
- *Consistent*: For any references `x` and `y`, multiple invocations of `x.equals(y)` will always return the same result, provided the objects referenced by these references have not been modified to affect the equals comparison.
- `null` *comparison*: For any non-null reference `obj`, the call `obj.equals(null)` always returns `false`.

The general contract of the `equals()` method is defined between *objects of arbitrary classes*. Understanding its criteria is important for providing a proper implementation.

Reflexivity

This rule simply states that an object is equal to itself, regardless of how it is modified. It is easy to satisfy: The object passed as an argument and the current object are compared for object reference equality (`==`), the same as the default behavior of the `Object.equals()` method. If the references are aliases, the `equals()` implementation returns `true` .

```
if (this == argumentObj)
  return true;
// ...
```

The reflexivity test can usually be an operand of the short-circuit conditional-OR operator (`||`) in a `return` statement, where if the reflexivity test is `true` , other criteria for equality comparison are not evaluated:

Click here to view code image

```
return (this == argumentObj)
    || (/* Other criteria for equality. */);
```

Symmetry

The expression `x.equals(y)` invokes the `equals()` method on the object referenced by the reference `x` , whereas the expression `y.equals(x)` invokes the `equals()` method on the object referenced by the reference `y` . If `x.equals(y)` is `true` , then `y.equals(x)` must be `true` .

If the `equals()` methods invoked are in different classes, the classes must bilaterally agree whether their objects are equal or not. In other words, symmetry can be violated if the `equals()` method of a class makes unilateral decisions about which classes it will interoperate with, while the other classes are not aware of this. Avoiding interoperability with other (non-related) classes when implementing the `equals()` method is strongly recommended.

Transitivity

If two classes, `A` and `B` , have a bilateral agreement on their objects being equal, then this rule guarantees that one of them, say `B` , does not enter into an agreement with a third class `C` on its own. All classes involved must multilaterally abide by the terms of the contract.

A typical pitfall resulting in broken transitivity is when the `equals()` method in a subclass calls the `equals()` method of its superclass, as part of its equals comparison. The `equals()` method in the subclass can be implemented by code equivalent to the following line:

Click here to view code image

```
return super.equals(argumentObj) && compareSubclassSpecificAspects();
```

The idea is to compare only the subclass-specific aspects in the subclass `equals()` method and to use the superclass `equals()` method for comparing the superclass-specific aspects. However, this approach should be used with extreme caution. The problem lies in getting the equivalence contract fulfilled bilaterally between the superclass and the subclass `equals()` methods. Symmetry or transitivity can easily be broken.

If the superclass is abstract, using the superclass `equals()` method works well. There are no superclass objects for the subclass `equals()` method to consider. In addition, the superclass `equals()` method cannot be called directly by any clients other than subclasses. The subclass `equals()` method then has control of how the superclass `equals()` method is called. It can safely call the superclass `equals()` method to compare the superclass-specific aspects of subclass objects.

Consistency

This rule enforces that two objects that are equal (or non-equal) remain equal (or non-equal) as long as they are not modified. For mutable objects, the result of the equals comparison can change if one (or both) are modified between method invocations. However, for immutable objects, the result must always be the same. The `equals()` method should take into consideration whether the class implements immutable objects, and ensure that the consistency rule is not violated.

`null` *Comparison* This rule states that no object is equal to `null`. The contract calls for the `equals()` method to return `false`. The method must not throw an exception; that would be violating the contract. A check for this rule is necessary in the implementation. Typically, the argument object is explicitly compared with the `null` value:

```
if (argumentObj == null)
  return false;
```

In many cases, it is preferable to use the `instanceof` pattern match operator. It determines whether the argument object is of the appropriate type and introduces a local

variable of the appropriate subtype that denotes the argument object. It is always `false` if its left operand is `null`:

Click here to view code image

```
if (!(argumentObj instanceof MyRefType other))
  return false;
// Local variable other can be used in further implementation.
```

Note that if the `instanceof` pattern match operator is `true`, the `if` condition is `false` so that the local variable `other` of `MyRefType` is introduced into the scope *after* the `if` statement. In fact, the `if` statement can often be eliminated if the fields are compared individually:

Click here to view code image

```
return (this == argumentObj)                          // Reflexivity test
    || (argumentObj instanceof MyRefType other        // null comparison & cast
        && /* Can use variable other to compare individual fields. */);
```

The `return` statement above will now also return `false` if the `instanceof` pattern match operator determines that `argumentObj` is `null` or if it is not of `MyRefType`. If the `instanceof` pattern match operator returns `true`, the reference `other` can be safely used to compare the individual fields by short-circuit evaluation of the `&&` operator (see **Example 14.3**).

Example 14.3 *Implementing the* `equals()` *Method*

Click here to view code image

```
import java.util.Objects;

// Overrides equals(), but not hashCode().
public class UsableVNO {

  private int release;
  private int revision;
  private int patch;

  public UsableVNO(int release, int revision, int patch) {
    this.release  = release;
    this.revision = revision;
    this.patch    = patch;
  }
```

```
  public int getRelease()  { return this.release;  }
  public int getRevision() { return this.revision; }
  public int getPatch()    { return this.patch;    }

  @Override public String toString() {
    return "(" + release + "." + revision + "." + patch + ")";
  }

  @Override public boolean equals(Object obj) {     // (1)
    return (this == obj)                            // (2)
        || (obj instanceof UsableVNO vno            // (3)
            && this.patch    == vno.patch           // (4)
            && this.revision == vno.revision
            && this.release  == vno.release);
  }
}
```

Checklist for Implementing the `equals()` Method

Example 14.3 shows an implementation of the `equals()` method for version numbers. Next, we provide a checklist for implementing the `equals()` method.

Method Overriding Signature

The method header is

Click here to view code image

```
public boolean equals(Object obj)          // (1)
```

The signature of the method requires that the argument passed is of the type `Object`. The following header will overload the method, not override it:

Click here to view code image

```
public boolean equals(MyRefType obj)      // Overloaded.
```

The compiler will not complain. Therefore, it is a good idea to use the `@Override` annotation. Calls to overloaded methods are resolved at compile time, depending on the type of the argument. Calls to overridden methods are resolved at runtime, depending on the type of the actual object referenced by the argument. Comparing the objects of the class `MyRefType` that *overloads* the `equals()` method for equivalence can give inconsistent results:

Click here to view code image

```
MyRefType ref1 = new MyRefType();
MyRefType ref2 = new MyRefType();
Object    ref3 = ref2;
boolean b1 = ref1.equals(ref2);    // True. Calls equals() in MyRefType.
boolean b2 = ref1.equals(ref3);    // Always false. Calls equals() in Object.
```

However, if the `equals()` method is overridden correctly, only the overriding method in `MyRefType` is called. A class can provide both implementations, but the `equals()` methods must be consistent. However, there must be a legitimate reason to overload the `equals()` method and this practice is not recommended.

Reflexivity Test

This is usually the first test performed in the `equals()` method, avoiding further computation if the test is `true`. The `equals()` method in **Example 14.3** does this test at (2):

Click here to view code image

```
return (this == obj)                                    // (2)
    || (/* */);
```

Correct Argument Type

The `equals()` method of the `UsableVNO` class in **Example 14.3** checks the type of the argument object and assigns its reference value to the local variable `vno` at (3) using the `instanceof` pattern match operator:

Click here to view code image

```
return (this == obj)                                    // (2)
    || (obj instanceof UsableVNO vno                    // (3)
        && /* Compare individual fields */);
```

This code also does the `null` comparison correctly, returning `false` if the argument `obj` has the value `null`.

The `instanceof` operator will also return `true` if the argument `obj` denotes a *subclass object* of the class `UsableVNO`. If the class is `final`, this issue does not arise—

there are no subclass objects. The following test can be explicitly performed in order to exclude all other objects, including subclass objects:

Click here to view code image

```
if ((obj == null) || (obj.getClass() != this.getClass()))
  return false;
```

The condition in the `if` statement first performs the `null` comparison. The expression `(obj.getClass() != this.getClass())` determines whether the classes of the two objects have the same runtime object representing them. If this is the case, the objects are instances of the same class.

Field Comparisons

Equivalence comparison involves comparing relevant fields from both objects to determine if their logical states match. For fields that are of primitive data types, their primitive values can be compared. Instances of the class `UsableVNO` in **Example 14.3** have fields of primitive data types only. Values of corresponding fields can be compared to test for equality between two `UsableVNO` objects, as shown at (4):

Click here to view code image

```
return (this == obj)                                    // (2)
    || (obj instanceof UsableVNO vno                    // (3)
        && this.patch    == vno.patch                   // (4)
        && this.revision == vno.revision
        && this.release  == vno.release);
```

If all field comparisons evaluate to `true`, the `equals()` method returns `true`.

For fields that are references, the objects referenced by the references can be compared. For example, if the `UsableVNO` class declares a field called `productInfo`, which is a reference, the following expression can be used:

Click here to view code image

```
(this.productInfo != null && this.productInfo.equals(vno.productInfo)))
```

In order to avoid a `NullPointerException` being thrown, the `equals()` method is not invoked if the `this.productInfo` reference is `null`. However, the `Objects.equals()` utility method makes this comparison much simpler, as it pro-

vides `null`-safe object value equality operation, only invoking the `equals()` method on the first `productInfo` field, if both arguments are not `null`:

Click here to view code image

```
Objects.equals(this.productInfo, vno.productInfo)
```

Exact comparison of floating-point values should not be done directly on the values, but on the integer values obtained from their bit patterns (see static methods `Float.floatToIntBits()` and `Double.doubleToLongBits()` in the Java SE Platform API documentation). This technique eliminates certain anomalies in floating-point comparisons that involve a NaN value or a negative zero (see also the `equals()` method in `Float` and `Double` classes).

Only fields that have significance for the equivalence relation should be considered. Derived fields, whose computation is dependent on other field values in the object, might be redundant to include. Including only the significant fields may be prudent. Computing the equivalence relation should be deterministic; therefore, the `equals()` method should not depend on unreliable resources, such as network access.

The order in which the comparisons of the significant fields are carried out can influence the performance of the equals comparison. Fields that are most likely to differ should be compared as early as possible, in order to short-circuit the computation. In our example, patch numbers evolve faster than revision numbers, which, in turn, evolve faster than release numbers. This order is reflected in the `return` statement at (4) in **Example 14.3**.

Above all, an implementation of the `equals()` method must ensure that the equivalence relation is fulfilled.

Example 14.4 is a client that uses the class `UsableVNO` from **Example 14.3**. This client runs the same tests as the client in **Example 14.2**. The difference is that the class `UsableVNO` overrides the `equals()` method.

Example 14.4 *Implications of Implementing the* `equals()` *Method*

Click here to view code image

```
import static java.lang.System.out;
import java.util.*;

public class TestUsableVNO {
  public static void main(String[] args) {
```

```
    // Print name of version number class:
    out.println(UsableVNO.class);

    // Three individual version numbers.
    UsableVNO svno1 = new UsableVNO(9,1,1);                              // (1)
    UsableVNO svno2 = new UsableVNO(9,1,1);                              // (2)
    UsableVNO svno3 = new UsableVNO(6,6,6);                              // (3)

    // An array of version numbers.
    UsableVNO[] versions =  new UsableVNO[] {                            // (4)
        new UsableVNO( 3,49, 1), new UsableVNO( 8,19,81),
        new UsableVNO( 2,48,28), new UsableVNO(10,23,78),
        new UsableVNO( 9, 1, 1)};

    out.printf ("  svno1: %s, svno2: %s, svno3: %s%n", svno1, svno2, svno3);
    out.println("Test object reference equality:");                      // (5)
    out.println("  svno1 == svno2:      " + (svno1 == svno2));           // (6)
    out.println("  svno1 == svno3:      " + (svno1 == svno3));           // (7)
    out.println("Test object value equality:");
    out.println("  svno1.equals(svno2): " + svno1.equals(svno2));         // (8)
    out.println("  svno1.equals(svno3): " + svno1.equals(svno3));         // (9)
    out.println();

    // Search key:
    UsableVNO searchKey = new UsableVNO(9,1,1);                          // (10)

    // Searching in an array:
    boolean found = false;
    for (UsableVNO version : versions) {
      found = searchKey.equals(version);                                 // (11)
      if (found) break;
    }
    out.println("Array: " + Arrays.toString(versions));                  // (12)
    out.printf("  Search key %s found in array:    %s%n%n",              // (13)
                searchKey, found);

    // Searching in a list:
    List<UsableVNO> vnoList = Arrays.asList(versions);                   // (14)
    out.println("List:  " + vnoList);
    out.printf("  Search key %s contained in list: %s%n%n", searchKey,
                vnoList.contains(searchKey));                            // (15)
  }
}
```

Output from the program:

Click here to view code image

```
class UsableVNO
  svno1: (9.1.1), svno2: (9.1.1), svno3: (6.6.6)
Test object reference equality:
  svno1 == svno2:      false
  svno1 == svno3:      false
Test object value equality:
  svno1.equals(svno2): true
  svno1.equals(svno3): false

Array: [(3.49.1), (8.19.81), (2.48.28), (10.23.78), (9.1.1)]
  Search key (9.1.1) found in array:    true

List:  [(3.49.1), (8.19.81), (2.48.28), (10.23.78), (9.1.1)]
  Search key (9.1.1) contained in list: true
```

The output from the program shows that object value equality is compared correctly. Object value equality is now based on identical states, as defined by the `equals()` method.

The search for a `UsableVNO` object in an array or a list of `UsableVNO` objects is successful, since the equals comparison is based on the states of the objects and not on their reference values.

Next, we look at how to fix the version numbers so that they can be used for searching in sets and maps.

14.3 Implementing the `hashCode()` Method

Hashing is an efficient technique for storing and retrieving data. A common hashing scheme uses an array where each element is a list of items. The array elements are called *buckets*. Operations in a hashing scheme involve computing an array index from an item. Converting an item to its array index is done by a *hash function*. The array index returned by the hash function is called the *hash code* of the item— it is also called the *hash value* of the item. The hash code identifies a particular bucket.

Storing an item involves the following steps:

1. Hash the item to determine the bucket.
2. If the item does not match an item already in the bucket, it is stored in the bucket.

Note that no duplicate items are stored. A lookup for an item is also a two-step process:

1. Hash the item to determine the bucket.

2. If the item matches an item in the bucket, the item is present; otherwise, it is not.

Different items can hash to the same bucket, meaning that the hash function returns the same hash code for these items. This condition is called a *collision*. The list maintained by a bucket contains the items that hash to the bucket.

The hash code of an item only identifies the bucket. Finding an item in the bucket entails a search, and requires an equality function to compare items. The items maintained in a hash-based storage scheme must, therefore, provide two essential functions: a hash function and an equality function.

The performance of a hashing scheme is largely affected by how well the hash function distributes a collection of items over the available buckets. A hash function should not be biased toward any particular hash codes. An ideal hash function produces a uniform distribution of hash codes for a collection of items across all possible hash codes. Such a hash function is not an easy task to design. Fortunately, heuristics exist for constructing adequate hash functions.

Here we mention two abstract data types that can be implemented using a hashing scheme:

- A *set* is an abstract data type that stores an unordered collection of unique items. Items in a set are called *elements* and the element to search for in a set is referred to as the *search key*. The hashing is used to store and lookup an item in a set, as explained earlier in this section.
- A *map* (also called, a *hash table*) is an abstract data type that maintains its items as *key–value entries*. An *entry* associates a *key* with a *value*. The keys in a hash table are unique. The hashing is done on a search key to provide efficient lookup of the entry with the associated value. Matching the search key with a key in an entry determines the value.

If objects of a class are to be maintained in hash-based collections and maps provided by the `java.util` package (see **Table 15.2**, **p. 788**), the class must provide appropriate implementations of the following methods from the `Object` class:

- A `hashCode()` method that produces hash codes for the objects
- An `equals()` method that tests objects for object value equality

```
int hashCode()
```

When storing objects in hash tables, this method can be used to get a hash code for an object. This method tries to return distinct integers for distinct objects as their default hash code. The `hashCode()` method is usually overridden by a class, as in the case of the wrapper classes and the `String` class.

As a general rule for implementing these methods, *a class that overrides the* `equals()` *method must override the* `hashCode()` *method.* Consequences of not doing so are illustrated in **Example 14.5** using the class `UsableVNO` that does not override the `hashCode()` method. Elements of this class are used as elements in a set that uses the `hashCode()` method of its elements to maintain them in the set. The output from the program shows that a set with the following elements is created:

Click here to view code image

```
Set: [(8.19.81), (2.48.28), (3.49.1), (9.1.1), (10.23.78)]
```

The `hashCode()` method from the `Object` class is not overridden by the `UsableVNO` class and is, therefore, used to compute the hash codes of the elements. This method returns the memory address of the object as the default hash code. The attempt to find the search key `(9.1.1)` in the set is unsuccessful:

Click here to view code image

```
Search key (9.1.1) contained in set: false
```

The output from the program shows the hash codes assigned by this method to the search key and the elements in the set:

Click here to view code image

```
Search key (9.1.1) has hash code: 2036368507
Hash codes for the elements:
    (3.49.1): 1705929636
   (8.19.81): 1221555852
   (2.48.28): 1509514333
  (10.23.78): 1556956098
     (9.1.1): 1252585652
```

The hash codes of two objects, which are equal according to the `equals()` method of the class `UsableVNO`, are not equal according to the `hashCode()` method of the `Object` class. Therefore, the version number `(9.1.1)` in the set has a different hash

code than the search key `(9.1.1)`. These objects hash to different buckets. The lookup for the search key is done in one bucket and does not find the element `(9.1.1)`, which is to be found in a completely different bucket. Just overriding the `equals()` method is not enough. The class `UsableVNO` violates the key tenet of the `hashCode()` contract: *Equal objects must produce equal hash codes.*

Example 14.5 *Implications of Not Overriding the* `Object.hashCode()` *Method*

Click here to view code image

```
import static java.lang.System.out;
import java.util.*;

public class TestUsableVNO2 {
  public static void main(String[] args) {
    // Print name of version number class:
    out.println(UsableVNO.class);

    // An array of version numbers.
    UsableVNO[] versions =  new UsableVNO[] {                          // (1)
        new UsableVNO( 3,49, 1), new UsableVNO( 8,19,81),
        new UsableVNO( 2,48,28), new UsableVNO(10,23,78),
        new UsableVNO( 9, 1, 1)};

    // Search key:
    UsableVNO searchKey = new UsableVNO(9,1,1);                        // (2)

    // Create a list:
    List<UsableVNO> vnoList = Arrays.asList(versions);                 // (3)

    // Searching in a set:
    Set<UsableVNO> vnoSet = new HashSet<>(vnoList);                    // (4)
    out.println("Set: " + vnoSet);
    out.printf("Search key %s contained in set:  %s%n", searchKey,
                vnoSet.contains(searchKey));                           // (5)
    out.println();

    // Search key and its hash code:
    out.printf("Search key %s has hash code: %d%n", searchKey,
               searchKey.hashCode());                                  // (6)

    // Hash values for elements:
    out.println("Hash codes for the elements:");
    for (UsableVNO element : versions) {                               // (7)
      out.printf("  %10s: %s%n", element, element.hashCode());
    }
```

```
  }
}
```

Output from the program:

Click here to view code image

```
class UsableVNO
Set: [(8.19.81), (2.48.28), (3.49.1), (9.1.1), (10.23.78)]
Search key (9.1.1) has hash code: 2036368507
Hash codes for the elements:
    (3.49.1): 1705929636
   (8.19.81): 1221555852
   (2.48.28): 1509514333
  (10.23.78): 1556956098
     (9.1.1): 1252585652
```

General Contract of the `hashCode()` Method

The general contract of the `hashCode()` method stipulates the following:

- *Consistency during execution is a necessary prerequisite*: Multiple invocations of the `hashCode()` method on an object must consistently return the same hash code during the execution of an application, provided the object is not modified to affect the result returned by the `equals()` method. The hash code need not remain consistent across different executions of the application. This means that using a pseudorandom number generator to produce hash codes is not a valid strategy.
- *Object value equality implies hash code equality*: If two objects are equal according to the `equals()` method, then the `hashCode()` method must produce the same hash code for these objects. This tenet ties in with the general contract of the `equals()` method.
- *Object value inequality places no restrictions on the hash code*: If two objects are unequal according to the `equals()` method, then the `hashCode()` method need not produce distinct hash codes for these objects. However, it is strongly recommended that the `hashCode()` method produce unequal hash codes for unequal objects.

Note that the hash code contract does not imply that objects with equal hash codes are equal. Not producing unequal hash codes for unequal objects can have an adverse effect on performance, as unequal objects with the same hash code will hash to the same bucket.

Heuristics for Implementing the `hashCode()` Method

In **Example 14.6**, the computation of the hash code in the `hashCode()` method of the `ReliableVNO` class embodies heuristics that can produce fairly reasonable hash functions. The hash code is computed according to the following formula:

Click here to view code image

```
hashValue = 11 * 31^3 + release * 31^2 + revision * 31^1 + patch
```

Only the significant fields that have bearing on the `equals()` method are included. This ensures that objects that are equal according to the `equals()` method also have equal hash codes according to the `hashCode()` method.

Example 14.6 *Implementing the* `hashCode()` *Method*

Click here to view code image

```
import java.util.Objects;

// Overrides both equals() and hashCode().
public class ReliableVNO {

  private int release;
  private int revision;
  private int patch;

  public ReliableVNO(int release, int revision, int patch) {
    this.release  = release;
    this.revision = revision;
    this.patch    = patch;
  }

  public int getRelease()  { return this.release;  }
  public int getRevision() { return this.revision; }
  public int getPatch()    { return this.patch;    }

  @Override public String toString() {
    return "(" + release + "." + revision + "." + patch + ")";
  }

  @Override public boolean equals(Object obj) {                  // (1)
    return (this == obj)                                         // (2)
        || (obj instanceof ReliableVNO vno                       // (3)
            && this.patch    == vno.patch                        // (4)
            && this.revision == vno.revision
```

```
            && this.release  == vno.release);
  }

  @Override public int hashCode() {                                  // (5)
    int hashValue = 11;
    hashValue = 31 * hashValue + release;
    hashValue = 31 * hashValue + revision;
    hashValue = 31 * hashValue + patch;
    return hashValue;

//  return Objects.hash(this.release, this.revision, this.patch); // (6)
  }
}
```

The basic idea is to compute an `int` hash code `sfVal` for each significant field `sf`, and include an assignment of the form shown at (1) in the computation below:

Click here to view code image

```
public int hashCode() {
  int sfVal;
  int hashValue = 11;
  ...
  sfVal = ...   // Compute hash code for each significant field sf. See below.
  hashValue = 31 * hashValue + sfVal;   // (1)
  ...
  return hashValue;
}
```

This setup ensures that the result from incorporating a field value is used to calculate the contribution from the next field value.

Calculating the hash code `sfVal` for a significant field `sf` depends on the type of the field:

- Field `sf` is a `boolean`:

  ```
  sfVal = sf ? 0 : 1;
  ```

- Field `sf` is a `byte, char, short, or int`:

  ```
  sfVal = (int)sf;
  ```

- Field `sf` is a `long`:

Click here to view code image

```
sfVal = (int) (sf ^ (sf >>> 32));
```

- Field `sf` is a `float`:

```
sfVal = Float.floatToInt(sf);
```

- Field `sf` is a `double`:
Click here to view code image

```
long sfValTemp = Double.doubleToLong(sf);
sfVal = (int) (sfValTemp ^ (sfValTemp >>> 32));
```

- Field `sf` is a reference that denotes an object. Typically, the `hashCode()` method is invoked recursively if the `equals()` method is invoked recursively:
Click here to view code image

```
sfVal = (sf == null ? 0 : sf.hashCode());
```

- Field `sf` is an array. The contribution from each element is calculated similarly to a field.

The order in which the fields are incorporated into the hash code computation will influence the hash code. Fields whose values are derived from other fields can be excluded. There is no point in feeding the hash function with redundant information, since this is unlikely to improve the value distribution. Fields that are not significant for the `equals()` method must be excluded; otherwise, the `hashCode()`

method might end up contradicting the `equals()` method. As with the `equals()` method, data from unreliable resources (e.g., network access) should not be used, and inclusion of transient fields should be avoided.

A legal or correct hash function does not necessarily mean it is appropriate or efficient. The classic example of a legal but inefficient hash function is

```
public int hashCode() {
  return 1949;
}
```

All objects using this method are assigned to the same bucket. The hash table is then no better than a list. For the sake of efficiency, a hash function should strive to pro-

duce unequal hash codes for unequal objects.

For numeric wrapper types, the `hashCode()` implementation returns an `int` representation of the primitive value, converting the primitive value to an `int`, if necessary. The `Boolean` objects for the `boolean` literals `true` and `false` have specific hash codes, which are returned by the `hashCode()` method.

The `hashCode()` method of the `String` class returns a hash code that is the value of a polynomial whose variable has the value 31; the coefficients are the characters in the string, and the degree is the string length minus 1. For example, the hash code of the string `"abc"` is computed as follows:

Click here to view code image

```
hashValue = 'a' * 31^2 + 'b' * 31^1 + 'c' * 31^0 = 97 * 31 * 31 + 98 * 31 + 99 = 96354
```

For immutable objects, the hash code can be cached—that is, calculated once and returned whenever the `hashCode()` method is called.

However, the `hash()` method of the `Objects` class makes it very convenient to compute the hash code for an object with multiple fields. The procedure outlined above for calculating the hash code for a version number can be replaced by a single statement, as shown at (6) in **Example 14.6**. Each `int` field value is autoboxed in an `Integer` and passed in the call to the `Objects.hash()` method which returns a hash code based on the field values of the version number.

Click here to view code image

```
return Objects.hash(this.release, this.revision, this.patch); // (6)
```

The client in **Example 14.7** demonstrates searching for objects of the class `ReliableVNO` in a set and in a map. The `ReliableVNO` objects in the array `versions` are used as elements in the set and as keys in the map.

Output from the program in **Example 14.7** shows that the search key `(9.1.1)` and the element `(9.1.1)` in the set `vnoSet` have the same hash code. The search is successful. These objects hash to the same bucket. Therefore, the search for the element takes place in the right bucket. It finds the element `(9.1.1)` using the `equals()` method by successfully checking for equality between the search key `(9.1.1)` and the element `(9.1.1)` of the set.

Each entry in the map `versionStatistics` represents a version number (the *key*) and the number of downloads (the *value*) for that version number. Output from the program also shows that the key `(9.1.1)` of the entry `(9.1.1)=6000` in the map has the same hash code as the search key `(9.1.1)`. The search is successful.

However, we still cannot use objects of the class `ReliableVNO` in *sorted* sets and maps, as no criteria for comparing the version numbers have been defined.

Example 14.7 *Implications of Implementing the* `hashCode()` *Method*

Click here to view code image

```
import static java.lang.System.out;
import java.util.*;

public class TestReliableVNO {
  public static void main(String[] args) {
    // Print name of version number class:
    out.println(ReliableVNO.class);
    // An array of version numbers.
    ReliableVNO[] versions =  new ReliableVNO[] {                          // (1)
        new ReliableVNO( 3,49, 1), new ReliableVNO( 8,19,81),
        new ReliableVNO( 2,48,28), new ReliableVNO(10,23,78),
        new ReliableVNO( 9, 1, 1)};

    // Search key and its hash code:
    ReliableVNO searchKey = new ReliableVNO( 9, 1, 1);                     // (2)
    out.printf("Search key %s has hash code: %d%n", searchKey,
        searchKey.hashCode());                                             // (3)

    // Print hash values:
    out.println("Hash codes:");
    for (ReliableVNO element : versions) {                                 // (4)
      out.printf("  %10s: %s%n", element, element.hashCode());
    }
    out.println();

    // Searching in a set:
    Set<ReliableVNO> vnoSet = new HashSet<>(Arrays.asList(versions));      // (5)
    out.println("Set: " + vnoSet);
    out.printf("Search key %s contained in set: %s%n%n", searchKey,
                vnoSet.contains(searchKey));                               // (6)

    // Searching in a map:
    Map<ReliableVNO, Integer> versionStatistics = new HashMap<>();         // (7)
    versionStatistics.put(versions[0], 2000);
    versionStatistics.put(versions[1], 3000);
```

```
        versionStatistics.put(versions[2], 4000);
        versionStatistics.put(versions[3], 5000);
        versionStatistics.put(versions[4], 6000);
        out.println("Map: " + versionStatistics);                    // (8)
        out.printf("Search key %s contained in map: %s%n", searchKey,
                   versionStatistics.containsKey(searchKey));       // (9)
    }
}
```

Output from the program:

Click here to view code image

```
class ReliableVNO
Search key (9.1.1) has hash code: 336382
Hash codes:
    (3.49.1): 332104

   (8.19.81): 336059
   (2.48.28): 331139
  (10.23.78): 338102
     (9.1.1): 336382

Set: [(10.23.78), (2.48.28), (9.1.1), (3.49.1), (8.19.81)]
Search key (9.1.1) contained in set: true

Map: {(10.23.78)=5000, (2.48.28)=4000, (9.1.1)=6000, (3.49.1)=2000,
      (8.19.81)=3000}
Search key (9.1.1) contained in map: true
```

14.4 Implementing the `java.lang.Comparable<E>` Interface

In order to sort the objects of a class, it should be possible to *compare* the objects. A *total ordering* allows objects to be compared. The criteria used to do the comparison depend on what is meaningful for the class. For example, comparison of `Integer` objects is based on the `int` value in an `Integer` object, whereas comparison of `String` objects is based on the Unicode value of the characters comprising the string in a `String` object. There can be more than one total ordering to compare the objects of a class. For example, another total ordering for `String` objects can be case insensitive—that is, treats uppercase characters as lowercase when comparing `String` objects.

The total ordering for objects of a class that is designated as the *default* ordering is called the *natural ordering*. A class defines the natural ordering for its object by implementing the `java.lang.Comparable<E>` generic interface. Other total orderings can

be defined by providing a *comparator* that implements the `java.util.Comparator<E>` generic interface.

We will look at the two generic interfaces `Comparable<E>` and `Comparator<E>` for comparing objects in the rest of this chapter. The `Comparable<E>` interface is implemented by the objects of the class—in other words, the class provides the implementation for comparing its objects according to its natural ordering. The class usually leaves the implementation of a total ordering to its clients who can decide which total ordering is desirable.

The general contract for the `Comparable<E>` interface is defined by its only abstract method `compareTo()`, making this interface technically a functional interface. However, it is not annotated with the `@FunctionalInterface` annotation, and it is not intended to be implemented by lambda expressions.

```
int compareTo(E other)
```

Returns a negative integer, zero, or a positive integer if the current object is less than, equal to, or greater than the specified object, respectively, based on the natural ordering. It throws a `ClassCastException` if the reference value passed in the argument cannot be compared to the current object. It throws a `Null-PointerException` if the argument is `null`.

Many of the standard classes in the Java SE APIs, such as the primitive wrapper classes, enum types, `String`, `LocalDate`, `LocalDateTime`, `LocalTime`, and `File`, implement the `Comparable<E>` interface. Objects implementing this interface can be used as

- Elements in a sorted set
- Keys in a sorted map
- Elements in lists that are sorted using the `Collections.sort()` or `List.sort()` method
- Elements in arrays that are sorted using the overloaded `Arrays.sort()` methods

The natural ordering for `String` objects (and `Character` objects) is lexicographical ordering—that is, their comparison is based on the Unicode value of each corresponding character in the strings. Objects of the `String` and `Character` classes will be lexicographically maintained as elements in a sorted set, or as keys in a sorted map that uses their natural ordering.

The natural ordering for objects of a numerical wrapper class is in ascending order of the values of the corresponding numerical primitive type. As elements in a sorted set or as keys in a sorted map that uses their natural ordering, the objects will be maintained in ascending order.

According to the natural ordering for objects of the `Boolean` class, a `Boolean` object representing the value `false` is less than a `Boolean` object representing the value `true`.

An implementation of the `compareTo()` method for the objects of a class should meet the following criteria:

- For any two objects of the class, if the first object is *less than*, *equal to*, or *greater than* the second object, then the second object must be *greater than*, *equal to*, or *less than* the first object, respectively—that is, the comparison is *anti-symmetric*.
- All three comparison relations (*less than*, *equal to*, *greater than*) embodied in the `compareTo()` method must be *transitive*. For example, for any objects `obj1`, `obj2`, and `obj3` of a class, if `obj1.compareTo(obj2) > 0` and `obj2.compareTo(obj3) > 0`, then `obj1.compareTo(obj3) > 0`.
- For any two objects of the class that compare as equal, the `compareTo()` method must return the same result if these two objects are compared with any other object—that is, the comparison is *congruent*.
- The `compareTo()` method is strongly recommended (but not required) to be *consistent with equals*—that is, `(obj1.compareTo(obj2) == 0) == (obj1.equals(obj2))` is `true`. This is recommended if the objects will be maintained in sorted sets or sorted maps.

The magnitude of non-zero values returned by the `compareTo()` method is immaterial; the sign indicates the result of the comparison. The general contract of the `compareTo()` method augments the general contract of the `equals()` method, providing a natural ordering of the compared objects. The equality test of the `compareTo()` method has the same provisions as that of the `equals()` method.

Implementing the `compareTo()` method is not much different from implementing the `equals()` method. In fact, given that the functionality of the `equals()` method is a subset of the functionality of the `compareTo()` method, the `equals()` implementation can call the `compareTo()` method. This guarantees that the two methods are always consistent with each other.

Click here to view code image

```
@Override public boolean equals(Object obj) {
  return (this == obj)
```

```
      || (obj instanceof Whatever other
          && this.compareTo(other) == 0);
  }
```

Example 14.8 *Implementing the* `compareTo()` *Method of the* `Comparable<E>` *Interface*

Click here to view code image

```
import java.util.Comparator;
import java.util.Objects;

public final class VersionNumber implements Comparable<VersionNumber> {

  private final int release;
  private final int revision;
  private final int patch;

  public VersionNumber(int release, int revision, int patch) {
    this.release  = release;
    this.revision = revision;
    this.patch    = patch;
  }

  public int getRelease()  { return this.release;  }
  public int getRevision() { return this.revision; }
  public int getPatch()    { return this.patch;    }

  @Override public String toString() {
    return "(" + release + "." + revision + "." + patch + ")";
  }

  @Override public boolean equals(Object obj) {                    // (1)
    return (this == obj)                                           // (2)
        || (obj instanceof VersionNumber vno                       // (3)
            && this.patch    == vno.patch
            && this.revision == vno.revision
            && this.release  == vno.release);
  }

  @Override public int hashCode() {
    return Objects.hash(this.release, this.revision, this.patch);  // (4)
  }

  @Override public int compareTo(VersionNumber vno) {              // (5)
    // Compare the release numbers.                                   (6)
    if (this.release != vno.release)
      return Integer.compare(this.release, vno.release);
```

```
    // Release numbers are equal,                                    (7)
    // must compare revision numbers.
    if (this.revision != vno.revision)
      return Integer.compare(this.revision, vno.revision);

    // Release and revision numbers are equal,                       (8)
    // patch numbers determine the ordering.
    return Integer.compare(this.patch, vno.patch);
  }

  /*
  @Override public int compareTo(VersionNumber vno) {              // (9a)
    return Comparator.comparingInt(VersionNumber::getRelease)      // (10a)
                     .thenComparingInt(VersionNumber::getRevision) // (11a)
                     .thenComparingInt(VersionNumber::getPatch)    // (12a)
                     .compare(this, vno);                          // (13a)
  }
  */
}
```

A `compareTo()` method is seldom implemented to interoperate with objects of other classes. For example, this is the case for primitive wrapper classes and the `String` class. The calls to the `compareTo()` method in the last three statements below all result in a compile-time error.

Click here to view code image

```
Integer iRef = 10;
Double dRef = 3.14;
String str = "ten";
StringBuilder sb = new StringBuilder("ten");
boolean b1 = iRef.compareTo(str) == 0 ;  // compareTo(Integer) not applicable to
                                         // arguments (String).
boolean b2 = dRef.compareTo(iRef) > 0;   // compareTo(Integer) not applicable to
                                         // arguments (Double).
boolean b3 = sb.compareTo(str) == 0; // compareTo(StringBuilder) not applicable to
                                     // arguments (String).
```

A straightforward implementation of the `compareTo()` method for version numbers is shown in **Example 14.8**. Note the specification of the `implements` clause in the class header. By parameterizing the `Comparable<E>` interface with the `VersionNumber` type, the class declaration explicitly excludes comparison with objects of other types. Only `VersionNumber`s can be compared.

Click here to view code image

```
public final class VersionNumber implements Comparable<VersionNumber> {
  ...
  @Override public int compareTo(VersionNumber vno) {                // (5)
  ...
  }
  ...
}
```

The signature of the `compareTo()` method is `compareTo(VersionNumber)`. In order to maintain backward compatibility with non-generic code, the compiler inserts the following *bridge method* with the signature `compareTo(Object)` into the class (§11.11, p. 615).

Click here to view code image

```
public int compareTo(Object obj) {   // NOT A GOOD IDEA TO RELY ON THIS METHOD!
  return this.compareTo((VersionNumber) obj);
}
```

In an implementation of the `compareTo()` method, the fields are compared with the most significant field first and the least significant field last. In the case of the version numbers, the release numbers are compared first, followed by the revision numbers, with the patch numbers being compared last. Note that the next lesser significant fields are only compared if the comparison of the previous higher significant fields yielded equality. Inequality between corresponding significant fields short-circuits the computation. If all significant fields are equal, a zero will be returned. This approach is shown in the implementation of the `compareTo()` method at (5) through (8) in **Example 14.8**.

Comparison of integer values in fields can be optimized. In the code for comparing the release numbers at (5) in **Example 14.8**, we have relied on implicit use of the `Integer.compareTo()` method called by the `Integer.compare()` method, and auto-boxing of the `int` field values:

Click here to view code image

```
if (this.release != vno.release)
  return Integer.compare(this.release, vno.release);
// Next field comparison
```

The code above can be replaced by the following code for doing the comparison, which relies on the difference between `int` values:

Click here to view code image

```
int releaseDiff = release - vno.release;
if (releaseDiff != 0)
  return releaseDiff;
// Next field comparison
```

However, this code can break if the difference is a value not in the range of the `int` type.

Significant fields with non-`boolean` primitive values are normally compared using the relational operators `<` and `>`. For comparing significant fields denoting constituent objects, the main options are to either invoke the `compareTo()` method on them, or use a comparator.

A more compact and elegant implementation of the `compareTo()` method for version numbers is shown at (9a) in **Example 14.8** that exclusively uses methods of the `Comparator<E>` interface (**p. 769**). The `compareTo()` method implementation below is an equivalent version of the `compareTo()` method at (9a) in **Example 14.8**, where the method chaining has been split up and the method references replaced with equivalent lambda expressions.

Click here to view code image

```
@Override public int compareTo(VersionNumber vno) {                // (9b)
  Comparator<VersionNumber> c1
      = Comparator.comparingInt(vn -> vn.getRelease());            // (10b)
  Comparator<VersionNumber> c2
      = c1.thenComparingInt(vn -> vn.getRevision());               // (11b)
  Comparator<VersionNumber> c3
      = c2.thenComparingInt(vn -> vn.getPatch());                  // (12b)
//return c3.compare(this, vno);                                    // (13b)
  return Objects.compare(this, vno, c3);                           // (13c)
}
```

The method implementations at (9a) and (9b) essentially use a *conditional comparator* that applies its constituent comparators conditionally in the order in which it is composed. The lambda expressions specified as arguments in the method calls above at (10b), (11b), and (12b) will extract the `int` value of a particular field in a version number when the expression is executed.

- The call to the `static` method `comparingInt()` at (10b) returns a `Comparator` that compares the version numbers by the `int` value of their `release` field.

- The call to the `default` method `thenComparingInt()` at (11b) on `Comparator c1` returns a composed `Comparator` that first compares version numbers using `Comparator c1`, and if the comparison result is `0`, compares them by the `revision` field.
- Analogous to (11b), the call to the `default` method `thenComparingInt()` at (12b) on `Comparator c2` returns a composed `Comparator` that first compares version numbers using `Comparator c2`, and if the comparison result is `0`, compares them by the `patch` field.
- It is the call to the functional method `compare()` at (13b) on the composed `Comparator c3` that does the actual comparison on the current version number (`this`) and the argument object (`vno`) according to the natural ordering of the `int` values of their fields, and where the order of the individual field comparisons is given by the composed `Comparator c3`.
  Equivalently, the `return` statement at (13b) can be written as (13c), where the convenience method `Objects.compare()` takes as arguments the current version number (`this`), the argument object (`vno`), and the composed `Comparator c3`. The `Objects.compare()` method calls the functional method `compare()` on the composed `Comparator c3`, passing it the current version number (`this`) and the argument object (`vno`).

What is different about this implementation of version numbers is that the class `VersionNumber` now overrides both the `equals()` and the `hashCode()` methods, and implements the `compareTo()` method of the parameterized `Comparable<VersionNumber>` interface. In addition, the `compareTo()` method is consistent with the `equals()` method. Following general class design principles, the class has been declared `final` so that it cannot be extended.

Example 14.9 *Implications of Implementing the* `compareTo()` *Method*

Click here to view code image

```
import static java.lang.System.out;
import java.util.*;

public class TestVersionNumber {
  public static void main(String[] args) {
    // Print name of version number class:
    out.println(VersionNumber.class);

    // Create an unsorted array of version numbers:
    VersionNumber[] versions =  new VersionNumber[] {                    // (1)
        new VersionNumber( 3,49, 1), new VersionNumber( 8,19,81),
        new VersionNumber( 2,48,28), new VersionNumber(10,23,78),
        new VersionNumber( 9, 1, 1)};
```

```
        out.println("Unsorted array: " + Arrays.toString(versions));

        // Create an unsorted list:
        List<VersionNumber> vnoList = Arrays.asList(versions);                  // (2)
        out.println("Unsorted list:  " + vnoList);

        // Create an unsorted map:
        Map<VersionNumber, Integer> versionStatistics = new HashMap<>();        // (3)
        versionStatistics.put(versions[0], 2000);
        versionStatistics.put(versions[1], 3000);
        versionStatistics.put(versions[2], 4000);
        versionStatistics.put(versions[3], 5000);
        versionStatistics.put(versions[4], 6000);
        out.println("Unsorted map: " + versionStatistics);

        // Sorted set:
        Set<VersionNumber> sortedSet = new TreeSet<>(vnoList);                  // (4)
        out.println("Sorted set: " + sortedSet);

        // Sorted map:
        Map<VersionNumber, Integer> sortedMap = new TreeMap<>(versionStatistics);//(5)
        out.println("Sorted map: " + sortedMap);

        // Sorted list:
        Collections.sort(vnoList);                                              // (6)
        out.println("Sorted list:    " + vnoList);

        // Searching in sorted list:
        VersionNumber searchKey = new VersionNumber( 9, 1, 1);                  // (7)
        int resultIndex = Collections.binarySearch(vnoList, searchKey);         // (8)
        out.printf("Binary search in sorted list found key %s at index: %d%n",
                    searchKey, resultIndex);

        // Sorted array:
        Arrays.sort(versions);                                                  // (9)
        out.println("Sorted array:   " + Arrays.toString(versions));

        // Searching in sorted array:
        int resultIndex2 = Arrays.binarySearch(versions, searchKey);            // (10)
        out.printf("Binary search in sorted array found key %s at index: %d%n",
                    searchKey, resultIndex2);
    }
}
```

Output from the program:

Click here to view code image

```
class VersionNumber
Unsorted array: [(3.49.1), (8.19.81), (2.48.28), (10.23.78), (9.1.1)]
Unsorted list:  [(3.49.1), (8.19.81), (2.48.28), (10.23.78), (9.1.1)]
Unsorted map: {(10.23.78)=5000, (3.49.1)=2000, (8.19.81)=3000, (9.1.1)=6000,
               (2.48.28)=4000}

Sorted set: [(2.48.28), (3.49.1), (8.19.81), (9.1.1), (10.23.78)]
Sorted map: {(2.48.28)=4000, (3.49.1)=2000, (8.19.81)=3000, (9.1.1)=6000,
             (10.23.78)=5000}

Sorted list:    [(2.48.28), (3.49.1), (8.19.81), (9.1.1), (10.23.78)]
Binary search in sorted list found key (9.1.1) at index: 3
Sorted array:   [(2.48.28), (3.49.1), (8.19.81), (9.1.1), (10.23.78)]
Binary search in sorted array found key (9.1.1) at index: 3
```

Example 14.9 is a client that uses the class `VersionNumber` from **Example 14.8**. Unlike previous attempts, the following code from **Example 14.9** demonstrates that `VersionNumber` objects can now be maintained in sorted sets and maps. A sorted set is created at (4) based on the unsorted list `vnoList`, and a sorted map is created at (5) based on the unsorted map `versionStatistics`.

Click here to view code image

```
Set<VersionNumber> sortedSet = new TreeSet<>(vnoList);                    // (4)
...
Map<VersionNumber, Integer> sortedMap = new TreeMap<>(versionStatistics); // (5)
```

The output from executing this code shows that the elements in the set and the keys of the map are sorted in the natural ordering for version numbers:

Click here to view code image

```
Sorted set: [(2.48.28), (3.49.1), (8.19.81), (9.1.1), (10.23.78)]
Sorted map: {(2.48.28)=4000, (3.49.1)=2000, (8.19.81)=3000, (9.1.1)=6000,
             (10.23.78)=5000}
```

By default, the class `TreeSet<E>` relies on its elements to implement the `equals()` method and the `compareTo()` method. The output from the program in **Example 14.9** shows that the `TreeSet<VersionNumber>` maintains its elements sorted in the natural ordering dictated by the `compareTo()` method. Analogously, the output from the program in **Example 14.9** shows that the `TreeMap<VersionNumber, Integer>` maintains its entries sorted on the keys which are in the natural ordering dictated by the `compareTo()` method.

We can run generic operations on collections of version numbers. Utility methods provided by the `Collections` and `Arrays` classes in the `java.util` package are discussed in §15.11, p. 856, and §15.12, p. 864, respectively.

The following code sorts the elements in the list `vnoList` created at (2) in Example 14.9 according to their natural order:

Click here to view code image

```
Collections.sort(vnoList);                                          // (6)
```

The output from executing this code shows that the elements in the list are indeed sorted in ascending order:

Click here to view code image

```
Unsorted list:  [(3.49.1), (8.19.81), (2.48.28), (10.23.78), (9.1.1)]
...
Sorted list:    [(2.48.28), (3.49.1), (8.19.81), (9.1.1), (10.23.78)]
```

A binary search can be run on this sorted list to find the index of the version number `(9.1.1)` referenced by the reference `searchKey` at (7) in Example 14.9:

Click here to view code image

```
VersionNumber searchKey = new VersionNumber( 9, 1, 1);              // (7)
int resultIndex = Collections.binarySearch(vnoList, searchKey);     // (8)
```

Natural ordering is assumed for the elements in the list. Executing the code prints the correct index of the search key in the sorted list:

Click here to view code image

```
Binary search in sorted list found key (9.1.1) at index: 3
```

Finally, the code in Example 14.9 sorts the elements in the array `versions` created at (1) according to their natural order:

Click here to view code image

```
Arrays.sort(versions);                                              // (9)
```

The output from executing this code shows that the elements in the array are sorted as expected in ascending order:

Click here to view code image

```
Unsorted array: [(3.49.1), (8.19.81), (2.48.28), (10.23.78), (9.1.1)]
...
Sorted array:   [(2.48.28), (3.49.1), (8.19.81), (9.1.1), (10.23.78)]
```

We can run a binary search on this sorted list:

Click here to view code image

```
int resultIndex2 = Arrays.binarySearch(versions, searchKey);          // (10)
```

Again, natural ordering is assumed for the elements in the array. Executing the code prints the correct index of the search key in the sorted array:

Click here to view code image

```
Binary search in sorted array found key (9.1.1) at index: 3
```

14.5 Implementing the `java.util.Comparator<E>` Interface

The `java.util.Comparator<E>` interface is a *functional interface* and is designated as such with the `@FunctionalInterface` annotation in the Java SE API documentation—in other words, it is intended to be implemented by lambda expressions. Apart from its sole abstract method `compare()`, it defines a number of useful `static` and `default` methods which are listed at the end of this section. Several of these methods are illustrated throughout this chapter.

Precise control of ordering can be achieved by creating a customized comparator that imposes a specific total ordering on the elements. All comparators implement the `Comparator<E>` interface, providing implementation for its abstract method:

```
int compare(E o1, E o2)
```

The `compare()` method returns a negative integer, zero, or a positive integer if the first object is less than, equal to, or greater than the second object, according to the total ordering—that is, its contract is equivalent to that of the `compareTo()` method of

the `Comparable<E>` interface. Since this method tests for equality, it is strongly recommended that its implementation does not contradict the semantics of the `equals()` method for the objects.

An alternative ordering to the default natural ordering can be specified by passing a `Comparator` to the constructor when the sorted set or map is created. The `Collections` and `Arrays` classes provide utility methods for sorting and searching, which also take a `Comparator` (§15.11, p. 856, and §15.12, p. 864).

Example 14.10 demonstrates the use of different comparators for strings. The program creates several empty sorted sets, using the `TreeSet<E>` class, where the comparator is passed to the constructor ((1b), (1c), (4)). Elements from the `words` array are added to each sorted set by the `Collections.addAll()` method, as at (6). A text representation of each sorted set is then printed, as at (7). The output shows the sort order in which the elements are maintained in the set.

Example 14.10 *Natural Ordering and Total Orderings*

Click here to view code image

```
import java.util.*;

public class ComparatorUsage {
  public static void main(String[] args) {

    String[] words = {"court", "Stuart", "report", "Resort",            // (1)
                      "assort", "support", "transport", "distort"};

    // Choice of comparator.
    Set<String> strSet1 = new TreeSet<>();            // (1a) Natural ordering
    Set<String> strSet2 = new TreeSet<>(String.CASE_INSENSITIVE_ORDER); // (1b)
    Set<String> strSet3 = new TreeSet<>(              // (1c) Rhyming ordering
      (String obj1, String obj2) -> {
        // Create reversed versions of the strings:                      (2)
        String reverseStr1 = new StringBuilder(obj1).reverse().toString();
        String reverseStr2 = new StringBuilder(obj2).reverse().toString();
        // Compare the reversed strings lexicographically.
        return reverseStr1.compareTo(reverseStr2);                    // (3)
      }
    );
    Set<String> strSet4 = new TreeSet<>(
        Comparator.comparingInt(String::length)       // (4) First length, then by
                  .thenComparing(Comparator.naturalOrder())// (5) natural ordering
    );
```

```
    // Add the elements from the words array to a set and print the set:
    Collections.addAll(strSet1, words);                                  // (6)
    System.out.println("Natural order:\n" + strSet1);                    // (7)
    Collections.addAll(strSet2, words);
    System.out.println("Case insensitive order:\n" + strSet2);
    Collections.addAll(strSet3, words);
    System.out.println("Rhyming order:\n" + strSet3);
    Collections.addAll(strSet4, words);
    System.out.println("Length, then natural order:\n" + strSet4);
  }
}
```

Output from the program:

Click here to view code image

```
Natural order:
[Resort, Stuart, assort, court, distort, report, support, transport]
Case insensitive order:
[assort, court, distort, report, Resort, Stuart, support, transport]
Rhyming order:
[Stuart, report, support, transport, Resort, assort, distort, court]
Length, then natural order:
[court, Resort, Stuart, assort, report, distort, support, transport]
```

The `String` class implements the `Comparable<E>` interface, providing an implementation of the `compareTo()` method. The `compareTo()` method defines the natural ordering for strings, which is lexicographical. The natural ordering is used to maintain the elements sorted lexicographically when the sorted set at (1a) is used. If we wish to maintain the strings in a different ordering, we need to provide a customized comparator.

The `String` class provides a static field (`CASE_INSENSITIVE_ORDER`) that denotes a comparator object with a `compare()` method that ignores the case when comparing strings lexicographically. This particular total ordering is used to maintain the elements sorted when the sorted set at (1b) is used. The comparator is passed as an argument to the set constructor. The output shows how the elements are maintained sorted in the set by this total ordering, which is a *case-insensitive ordering*.

We can create a string comparator that enforces *rhyming ordering* on the strings. In rhyming ordering, two strings are compared by examining their corresponding characters at each position in the two strings, starting with the characters in the *last* position. First the characters in the last position are compared, then those in the last but

one position, and so on. For example, given the two strings `"report"` and `"court"`, the last two characters in both strings are the same. Continuing toward the start of the two strings, the character `'o'` in the first string is less than the character `'u'` in the second string. According to rhyming ordering, the string `"report"` is less than the string `"court"`.

Comparing two strings according to rhyming ordering is equivalent to reversing the strings and comparing the reversed strings lexicographically. If we reverse the two strings `"report"` and `"court"`, the reversed string `"troper"` is lexicographically less than the reversed string `"truoc"`.

A rhyming ordering comparator is implemented by the lambda expression at (1c) in **Example 14.10**. The lambda expression first creates reversed versions of the strings passed as arguments. A reversed version of a string is created using a string builder, which is first reversed and then converted back to a string, as shown at (2). The `compareTo()` method call at (3) compares the reversed strings, as the lexicographical ordering for the reversed strings is equivalent to the rhyming ordering for the original strings. This particular total ordering is used to maintain the elements sorted when the sorted set at (1c) is used. The lambda expression is passed as an argument to the set constructor, and is executed when the `compare()` method of the `Comparator<String>` interface is called by the sorted set implementation. The output shows how the elements are maintained sorted in the set by this total ordering, which is *rhyming ordering*.

Finally, a conditional comparator is composed at (4) and (5) that first compares the strings by their length, and if the lengths are equal, it employs the natural ordering for strings. The output shows that the strings are first sorted according to their lengths, and strings with equal lengths are sorted according to their natural ordering.

Example 14.11 illustrates using a comparator that orders version numbers (**Example 14.8**, **p. 763**) according to their reverse natural ordering. The method `Comparator.reversedOrder()` readily returns a comparator that imposes the reverse of the natural ordering.

A list of version numbers is initialized at (2). This list is sorted using the reverse natural ordering at (3). A binary search is done in this list at (4). We have used the same comparator for the search as we did for the sorting, in order to obtain predictable results. Searching this list with natural ordering at (5) does not find the key.

Example 14.11 *Using a Comparator for Version Numbers*

Click here to view code image

```
import static java.lang.System.out;
import java.util.*;

public class UsingVersionNumberComparator {

  public static void main(String[] args) {
    VersionNumber[] versions = new VersionNumber[] {                    // (1)
        new VersionNumber(3, 49, 1), new VersionNumber(8, 19, 81),
        new VersionNumber(2, 48, 28), new VersionNumber(10, 23, 78),
        new VersionNumber(9, 1, 1) };

    List<VersionNumber> vnList = new ArrayList<>();
    Collections.addAll(vnList, versions);                               // (2)
    out.println("List before sorting:\n  " + vnList);
    Collections.sort(vnList, Comparator.reverseOrder());                // (3)
    out.println("List after sorting according to " +
                "reverse natural ordering:\n  " + vnList);

    VersionNumber searchKey = new VersionNumber(9, 1, 1);
    int resultIndex = Collections.binarySearch(vnList, searchKey,
                                               Comparator.reverseOrder());  // (4)
    out.printf("Binary search in list using reverse natural ordering"
             + " found key %s at index: %d%n", searchKey, resultIndex);

    resultIndex = Collections.binarySearch(vnList, searchKey);          // (5)
    out.printf("Binary search in list using natural ordering"
             + " found key %s at index: %d%n", searchKey, resultIndex);
  }
}
```

Output from the program:

Click here to view code image

```
List before sorting:
  [(3.49.1), (8.19.81), (2.48.28), (10.23.78), (9.1.1)]
List after sorting according to reverse natural ordering:
  [(10.23.78), (9.1.1), (8.19.81), (3.49.1), (2.48.28)]
Binary search in list using reverse natural ordering found key (9.1.1) at index: 1
Binary search in list using natural ordering found key (9.1.1) at index: -6
```

The `Comparator<E>` interface also provides many useful `static` and `default` methods, including composing conditional comparators that can compare on multiple fields. Reference is given in the description below to where the methods are used.

Click here to view code image

```
default Comparator<T> reversed()
static <T extends Comparable<? super T>> Comparator<T> naturalOrder()
static <T extends Comparable<? super T>> Comparator<T> reverseOrder()
```

The first method returns a comparator that imposes the reverse ordering of this comparator, equivalent to `(a, b) -> this.compare(b, a)`.

The second method returns a comparator that compares `Comparable` objects in natural order, equivalent to `(a, b) -> a.compareTo(b)`.

The third method returns a comparator that imposes the reverse of the natural ordering on `Comparable` objects, equivalent to `(a, b) -> b.compareTo(a)`.

See **Example 14.10**, **p. 770**, and **Example 14.11**, **p. 772**.

Click here to view code image

```
static <T> Comparator<T> nullsFirst(Comparator<? super T> cmp)
static <T> Comparator<T> nullsLast(Comparator<? super T> cmp)
```

Return a `null`-friendly comparator that considers `null` to be either less than non-`null` or greater than non-`null`, respectively. These are useful comparators for sorting or searching in collections and maps when `null`s are considered as actual values.

Click here to view code image

```
static <T, U> Comparator<T>
        comparing(Function<? super T,? extends U> func,
                  Comparator<? super U> cmp)
```

Returns a `Comparator<T>` that applies `func` to the two given elements and compares the results using the specified `Comparator cmp`. It is effectively equivalent to `(a, b) -> cmp.compare(func.apply(a), func.apply(b))`.

Click here to view code image

```
static <T, U extends Comparable<? super U>> Comparator<T>
        comparing(Function<? super T,? extends U> func)
```

Returns a `Comparator<T>` that applies `func` to the two given elements first, before comparing the results by natural ordering. It is effectively equivalent to `(a, b) -> func.apply(a).compareTo(func.apply(b))`.

Click here to view code image

```
static <T> Comparator<T>
        comparingPrimType(ToPrimTypeFunction<? super T> func)
```

Returns a `Comparator<T>` that applies `func` to the two given elements first, before comparing the primitive-value results.

PrimType is either an `Int`, `Long`, or `Double`.

See **Example 14.8**, **p. 763**, and **Example 14.10**, **p. 770**.

Click here to view code image

```
default Comparator<T>
        thenComparing(Comparator<? super T> cmp)
```

Returns a *conditional* comparator that first determines using `this Comparator` whether two given elements are equal. If they are equal, the elements are compared using the specified `Comparator cmp`. Effectively, this method first executes `this.compare(a, b)`, and then `cmp.compare(a, b)` if necessary.

See **Example 14.10**, **p. 770**.

Click here to view code image

```
default <U> Comparator<T>
        thenComparing(Function<? super T,? extends U> func,
                      Comparator<? super U> cmp)
```

Returns a conditional comparator that first determines using `this Comparator` whether two given elements are equal. If they are equal, it then applies `func` to the elements and the results are compared using the specified `Comparator cmp`. Effectively, this method first executes `this.compare(a, b)`, and then `cmp.compare (func.apply(a), func.apply(b))` if necessary.

Click here to view code image

```
default <U extends Comparable<? super U>> Comparator<T>
        thenComparing(Function<? super T,? extends U> func)
```

Returns a conditional comparator that first determines using `this Comparator` whether two given elements are equal. If they are equal, it then applies `func` to the elements and the results are compared by natural ordering. Effectively, this method first executes `this.compare(a, b)`, and then `func.apply(a).compareTo(func.apply(b))` if necessary.

Click here to view code image

```
default Comparator<T>
        thenComparingPrimType(ToPrimTypeFunction<? super T> func)
```

PrimType is either an `Int`, `Long`, or `Double`.

These primitive-type specialized methods return a conditional comparator that first determines, using `this Comparator`, if two given elements are equal. If they are equal, it then applies `func` to the two elements and the primitive-value results are compared.

See **Example 14.8**, **p. 763**, and **Example 14.10**, **p. 770**.

Review Questions

14.1 Which of the following statements are true about the `hashCode()` and `equals()` methods?

Select the two correct answers.

a. Two objects that are different according to the `equals()` method must have different hash codes.

b. Two objects that are equal according to the `equals()` method must have the same hash code.

c. Two objects that have the same hash code must be equal according to the `equals()` method.

d. Two objects that have different hash codes must be unequal according to the `equals()` method.

14.2 Given that the objects referenced by the parameters override the `equals()` and `hashCode()` methods appropriately, which return values are possible from the following method?

Click here to view code image

```
String func(Object x, Object y) {
  return (x == y) + " " + x.equals(y) + " " + (x.hashCode() == y.hashCode());
}
```

Select the four correct answers.

a. `"false false false"`

b. `"false false true"`

c. `"false true false"`

d. `"false true true"`

e. `"true false false"`

f. `"true false true"`

g. `"true true false"`

h. `"true true true"`

14.3 Which code, when inserted at (1), in the `equalsImpl()` method will fulfill the contract of the `equals()` method?

Click here to view code image

```
public class Pair {
  private int a, b;
  public Pair(int a, int b) {
    this.a = a;
    this.b = b;
  }

  public boolean equals(Object o) {
    return (this == o) || (o instanceof Pair) && equalsImpl((Pair) o);
```

```
  }

  private boolean equalsImpl(Pair o) {
    // (1) INSERT CODE HERE
  }
}
```

Select the three correct answers.

a. `return a == o.a || b == o.b;`

b. `return false;`

c. `return a >= o.a;`

d. `return a == o.a;`

e. `return a == o.a && b == o.b;`

14.4 Which code, when inserted at (1), will provide a correct implementation of the `hashCode()` method in the following program?

Click here to view code image

```
import java.util.*;
public class Measurement {
  private int count;
  private int accumulated;
  public Measurement() {}
  public void record(int v) {
    count++;
    accumulated += v;
  }
  public int average() {
    return accumulated/count;
  }
  public boolean equals(Object other) {
    if (this == other)
      return true;
    if (!(other instanceof Measurement))
      return false;
    Measurement o = (Measurement) other;
    if (count != 0 && o.count != 0)
      return average() == o.average();
    return count == o.count;
  }
```

```
  public int hashCode() {
    // (1) INSERT CODE HERE
  }
}
```

Select the two correct answers.

a. `return 31337;`

b. `return accumulated / count;`

c.

Click here to view code image

```
return (count << 16) ^ accumulated;
```

d. `return ~accumulated;`

e.

Click here to view code image

```
return count == 0 ? 0 : average();
```

14.5 Which statement is true about the following program?

Click here to view code image

```
import java.util.Comparator;
import java.util.Comparator;
class Person  implements Comparable<Person > {
  private String name;

  private int age;
  Person (String name, int age) { this.name = name; this.age = age; }

  public int compareTo(Person  p2) {
    Comparator<String> strCmp = Person.cmp();
    int status = strCmp.compare(this.name, p2.name);
    if (status == 0) {
      Comparator<Integer> intCmp = Person.cmp();
      status = intCmp.compare(this.age, p2.age);
    }
```

```
    return status;
  }

  public static <E extends Comparable<E>> Comparator<E> cmp() {
    return  (e1, e2) -> e2.compareTo(e1);
  }
}

public class Main {
  public static void main(String[] args) {
    Person  p1 = new Person ("Tom", 20);
    Person  p2 = new Person ("Dick", 30);
    Person  p3 = new Person ("Tom", 40);
    System.out.println((p1.compareTo(p2) < 0) + " " + (p1.compareTo(p3) < 0));
  }
}
```

Select the one correct answer.

a. The program will fail to compile.

b. The program will compile. When run, it will throw an exception.

c. The program will compile. When run, it will print `true false`.

d. The program will compile. When run, it will print `true true`.

e. The program will compile. When run, it will print `false false`.

f. The program will compile. When run, it will print `false true`.

14.6 Which method returns a `Comparator<E>` that is not equivalent to the other six?

Click here to view code image

```
import java.util.Comparator;
class CompDecls {

  public static <E extends Comparable<E>> Comparator<E> cmp1() {     // (1)
    return new Comparator<E>() {
      public int compare(E e1, E e2) { return e2.compareTo(e1); }
    };
  }

  public static <E extends Comparable<E>> Comparator<E> cmp2() {     // (2)
    return (e1, e2) -> e2.compareTo(e1);
```

```
  }

  public static <E extends Comparable<E>> Comparator<E> cmp3() {     // (3)
    return ((Comparator<E>)(e1, e2) -> e1.compareTo(e2)).reversed();
  }

  public static <E extends Comparable<E>> Comparator<E> cmp4() {     // (4)
    return Comparable::compareTo;
  }

  public static <E extends Comparable<E>> Comparator<E> cmp5() {     // (5)
    return ((Comparator<E>)Comparable::compareTo).reversed();
  }

  public static <E extends Comparable<E>> Comparator<E> cmp6() {     // (6)
    Comparator<E> cmp =  Comparable::compareTo;
    return cmp.reversed();
  }

  public static <E extends Comparable<E>> Comparator<E> cmp7() {     // (7)
    return Comparator.reverseOrder();
  }
}
```

Select the one correct answer.

a. (1)

b. (2)

c. (3)

d. (4)

e. (5)

f. (6)

g. (7)

14.7 Given the following code:

Click here to view code image

```
import java.util.*;
public class Test14RQ8 {
```

```
  public static void main(String[] args) {
    Integer[] values = {-42,15,-23,19,11,71};
    Arrays.sort(values, (v1, v2) -> v1.toString().compareTo(v2.toString()));
    System.out.println(Arrays.toString(values));
  }
}
```

What is the result?

Select the one correct answer.

a. `[71, 19, 15, 11, -23, -42]`

b. `[-23, -42, 11, 15, 19, 71]`

c. `[-42, -23, 11, 15, 19, 71]`

d. `[71, 19, 15, 11, -42, -23]`

e. `[-42, 15, -23, 19, 11, 71]`

f. The program will throw an exception at runtime.

g. The program will fail to compile.

14.8 Given the following code:

Click here to view code image

```
import java.util.*;
import java.util.function.*;
public class Album {
  private static List<Album> albums = new ArrayList<>();
  private String title;
  private Album(String title) { this.title = title; }
  public String toString() { return title; }
  public static void addAlbum(String title) {
    albums.add(new Album(title));
  }
  public static void sortAlbums(Comparator<Album> c) {
    albums.sort(c);
  }
  public static void processAlbums(Consumer<Album> c) {
    albums.forEach(c);
  }
}
```

and

Click here to view code image

```
public class Test14RQ11 {
  public static void main(String[] args) {
    Album.addAlbum("New Songs");
    Album.addAlbum("More Songs");
    Album.addAlbum("Greatest Hits");
    Album.addAlbum("Old Songs");
    Album.sortAlbums((a1, a2) -> a1.toString().length() - a2.toString().length());
    Album.processAlbums(a -> {
      System.out.print(a.toString() + " ");
    });
  }
}
```

What is the result?

Select the one correct answer.

a.

Click here to view code image

```
New Songs Old Songs More Songs Greatest Hits
```

b.

Click here to view code image

```
New Songs More Songs Greatest Hits Old Songs
```

c.

Click here to view code image

```
Greatest Hits More Songs Old Songs New Songs
```

d.

Click here to view code image

```
Old Songs Greatest Hits More Songs New Songs
```

e. The program will throw an exception at runtime.

f. The program will fail to compile.

14.9 Given the following code:

Click here to view code image

```
import java.util.Objects;
public class Album1 {
  private String title;
  public Album1(String title) { this.title = title; }
  public int hashCode() {
    int hash = 7;
    hash = 29 * hash + Objects.hashCode(this.title);
    return hash;
  }
  public boolean equals(Object obj) {
    if (this == obj) { return true; }
    if (obj == null) { return false; }
    if (getClass() != obj.getClass()) { return false; }
    final Album1 other = (Album1) obj;
    return Objects.equals(this.title, other.title);
  }
}
```

and

Click here to view code image

```
public class LP extends Album1 {
  public LP(String title) { super(title); }
}
```

and

Click here to view code image

```
public class Test14RQ9 {
  public static void main(String[] args) {
    Album1 a1 = new Album1("Some Music");
    Album1 a2 = new LP("Some Music");
```

```
      if (a1.equals(a2)) {
        System.out.println("Same album");
      } else {
         System.out.println("These are different albums");
      }
    }
  }
```

What is the result?

Select the one correct answer.

a. `Same album`

b. `These are different albums`

c. The program will throw an exception at runtime.

d. The program will fail to compile.

14.10 Which of the following statements are true about object ordering? Select the two correct answers.

a. Class `A` can implement the `Comparator<A>` interface to compare its instances to other objects.

b. Class `A` can implement the `Comparable<A>` interface to compare its instances to other objects.

c. The `Comparator<A>` interface is implemented to establish a natural ordering for objects of class `A`.

d. The `Comparable<A>` interface is implemented to establish a natural ordering for objects of class `A`.