Chapter 6
Comparing Objects

Advanced Topics in Java

Reference:

A Programmer's Guide to Java SE 8 Oracle Certified Professional (OCP),
Khalid A. Mughal,
Addison-Wesley Professional, *To be published in 2019*.

Khalid Azim Mughal
Associate Professor Emeritus
Department of Informatics
University of Bergen, Norway.
khalid.mughal@uib.no
http://www.ii.uib.no/~khalid

Version date: 2018-11-06

Overview

- Comparing Objects
- The equals() Method
- The hashCode() Method
- The Comparable<E> Interface
- The Comparator<E> Interface

The Object Class

- The majority of the non-`final` methods of the `Object` class are meant to be overridden.

`String toString()`

If a subclass does not override this method, it returns a textual representation of the object, which has the following format:

"*<name of the class>*@*<hash code value of object>*"

`boolean equals(Object obj)`

The `equals()` method in the `Object` class returns `true` only if the two references compared denote the same object, a.k.a. *object reference equality*.

```
int hashCode()                                    // §, p. 28.
```

When storing objects in hash tables, this method can be used to get a hash code for an object.

This value is guaranteed to be consistent during the execution of the program, provided the information used in the `equals()` comparisons on the object does not change.

- Convenient static methods of the `java.util.Objects` class—they are `null`-safe.

> `boolean equals(Object obj1, Object obj2)`
> Returns `true` if the arguments are equal to each other, otherwise `false`. Which 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)`.
>
> `int hash(Object... values)`
> Generates a hash code for a sequence of specified values.
>
> `int hashCode(Object obj)`
> Computes and returns the hash code of a non-`null` argument, otherwise returns `0` for a `null` argument.
>
> `<T> int compare(T t1, T t2, Comparator<? super T> cmp)`
> Returns `0` if the arguments are identical, otherwise calls `cmp.compare(t1, t2)`. It also returns `0` if both arguments are `null`.

Running example: *Version Numbers* **(VNO).**

- A version number (VNO) for a software product comprises three pieces of information:
  - a release number
  - a revision number
  - a patch number
- Ranking by ordering VNOs
  - 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.
- The static generic `test()` method in the `TestCaseVNO` class runs the tests.

Example 6.1 A Test Case for Version Numbers

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

import java.util.Arrays;
import java.util.Collections;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.TreeMap;
import java.util.TreeSet;

public class TestCaseVNO {
  /**  Type parameter N represents a class implementing a version number. */
  public static <N> void test(                                      // (1)
               N latest,                                            // (2a)
               N inShops,                                           // (2b)
               N older,                                             // (2c)
               N[] versions,                                        // (3)
               Integer[] downloads) {                               // (4)

    // Print the class name.
    out.println(latest.getClass());                                 // (5)

    // Various equality tests.
    out.println("Test object reference and value equality:");
```

```
out.printf ("    latest: %s, inShops: %s, older: %s%n",
                 latest, inShops, older);
out.println("    latest == inShops:      " + (latest == inShops));    // (6)
out.println("    latest.equals(inShops): "
            +    (latest.equals(inShops)));                           // (7)
out.println("    latest == older:        " + (latest == older));      // (8)
out.println("    latest.equals(older):   " + latest.equals(older));   // (9)

// Searching in an array:
N searchKey = inShops;                                                // (10)
boolean found = false;
for (N 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",
           searchKey, found);                                         // (13)

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

```
// Searching in a map:
Map<N, Integer> versionStatistics = new HashMap<>();                    // (16)
for (int i = 0; i < versions.length; i++) {                             // (17)
  versionStatistics.put(versions[i], downloads[i]);
}
out.println("Map: " + versionStatistics);                               // (18)
out.println("    Hash code for keys in the map:");
for (N version : versions) {                                            // (19)
  out.printf("    %10s: %s%n", version, version.hashCode());
}
out.printf("    Search key %s has hash code: %d%n", searchKey,
            searchKey.hashCode());                                      // (20)
out.printf("    Map contains search key %s: %s%n", searchKey,
           versionStatistics.containsKey(searchKey));                   // (21)

// Sorting collections and maps:
out.println("Sorted set:\n    " + (new TreeSet<>(vnoList)));            // (22)
out.println("Sorted map:\n    " +
            (new TreeMap<>(versionStatistics)));                        // (23)

out.println("List before sorting: " + vnoList);
Collections.sort(vnoList, null);                                        // (24)
out.println("List after sorting:  " + vnoList);

// Searching in sorted list:
```

```
    int resultIndex = Collections.binarySearch(vnoList, searchKey, null);// (25)
    out.printf("Binary search in list found key %s at index: %d%n",
                searchKey, resultIndex);
  }
}
```

- Possible output:

```
class VersionNumber
Test object reference and value equality:
    latest: (9.1.1), inShops: (9.1.1), older: (6.6.6)
    latest == inShops:      false
    latest.equals(inShops): true
    latest == older:        false
    latest.equals(older):   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
Map: {(10.23.78)=1010, (3.49.1)=245, (8.19.81)=786, (9.1.1)=123, (2.48.28)=54}
    Hash code for keys in the map:
      (3.49.1): 34194
     (8.19.81): 38149
     (2.48.28): 33229
    (10.23.78): 40192
```

```
        (9.1.1): 38472
    Search key (9.1.1) has hash code: 38472
    Map contains search key (9.1.1): true
```
Sorted set:

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

```
    {(2.48.28)=54, (3.49.1)=245, (8.19.81)=786, (9.1.1)=123, (10.23.78)=1010}
List before sorting: [(3.49.1), (8.19.81), (2.48.28), (10.23.78), (9.1.1)]
List after sorting:  [(2.48.28), (3.49.1), (8.19.81), (9.1.1), (10.23.78)]
```
Binary search in list found key (9.1.1) at index: 3

The `equals()` Method

- The `Object.equals()` method implements *object reference equality*.
  - Each instance of the class is only equal to itself.

Example 6.2 Not overriding the `equals()` *and the* `hashCode()` *Methods*

```
// Does not override equals() or hashCode().
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 + ")";
  }
}
```

Example 6.3 Testing the `equals()` *and the* `hashCode()` *Methods*

```
public class TestSimpleVNO {
  public static void main(String[] args) {
    // Three individual version numbers.
    SimpleVNO latest  = new SimpleVNO(9,1,1);                           // (1)
    SimpleVNO inShops = new SimpleVNO(9,1,1);                           // (2)
    SimpleVNO older   = new SimpleVNO(6,6,6);                           // (3)

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

    // An array with number of downloads.
    Integer[] downloads = {245, 786, 54,1010, 123};                     // (5)

    TestCaseVNO.test(latest, inShops, older, versions, downloads);      // (6)
  }
}
```

- Possible output from the program:

```
class SimpleVNO
```

```
Test object reference and value equality:
    latest: (9.1.1), inShops: (9.1.1), older: (6.6.6)
    latest == inShops:      false
    latest.equals(inShops): false
    latest == older:        false
    latest.equals(older):   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: false
List: [(3.49.1), (8.19.81), (2.48.28), (10.23.78), (9.1.1)]
    Search key (9.1.1) contained in list: false
Map: {(2.48.28)=54, (9.1.1)=123, (3.49.1)=245, (8.19.81)=786, (10.23.78)=1010}
    Hash code for keys in the map:
      (3.49.1): 31168322
     (8.19.81): 17225372
     (2.48.28): 5433634
    (10.23.78): 2430287
       (9.1.1): 17689166
    Search key (9.1.1) has hash code: 6585861
    Map contains search key (9.1.1): false
Exception in thread "main" java.lang.ClassCastException: SimpleVNO cannot be cast
to java.lang.Comparable
     ...
      at TestCaseVNO.test(TestCaseVNO.java:66)
```

```
at TestSimpleVNO.main(TestSimpleVNO.java:18)
```

Equivalence Relation for 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*.

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 argument and the current object are compared for *object reference equality* (==):

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

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.
  - Avoiding interoperability with other (non-related) classes.

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 usually has code equivalent to the following line:

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

– Symmetry or transitivity can easily be broken.

- If the superclass is abstract, using the superclass `equals()` method works well.

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.
  – 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.

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

or

```
if (!(argumentObj instanceof MyRefType))
  return false;
```

Example 6.4 Implementing the `equals()` Method

Example 6.5

```
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)
```

```
    if (obj == this)                                  // (2)
      return true;
    if (!(obj instanceof UsableVNO))                  // (3)
      return false;
    UsableVNO vno = (UsableVNO) obj;                  // (4)
    return vno.patch    == this.patch                 // (5a)
        && vno.revision == this.revision
        && vno.release  == this.release;

//  return Objects.equals(vno.patch, this.patch)    // (5b)
//      && Objects.equals(vno.revision, this.revision)
//      && Objects.equals(vno.release, this.release);
  }
}
```

A checklist for implementing the `equals()` method.

Method Overriding signature

- The method header is

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

- The following header will overload the method, not override it:

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

```
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.
```

Reflexivity Test

- The first test performed in the `equals()` method, avoiding further computation if the test is `true`.

```
if (obj == this)                                // (2)
  return true;
```

Correct Argument Type

- Checks the type of the argument object at (3), using the `instanceof` operator:

```
if (!(obj instanceof UsableVNO))                // (3)
  return false;
```

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

```
if ((obj == null) || (obj.getClass() != this.getClass()))   // (3a)
  return false;
```

Argument Casting

- The argument is only cast after checking that the cast will be successful.

```
UsableVNO vno = (UsableVNO) obj;                  // (4)
```

Field Comparisons

- Equivalence comparison involves comparing certain fields from both objects to determine if their logical states match.

```
return vno.patch    == this.patch               // (5a)
    && vno.revision == this.revision
    && vno.release  == this.release;
```

- For fields that are references, the objects referenced by the references can be compared.

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

is equivalent to

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

```
return Objects.equals(vno.patch, this.patch)    // (5b)
    && Objects.equals(vno.revision, this.revision)
    && Objects.equals(vno.release, this.release);
```

- 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 API documentation).
- Only fields that have significance for the equivalence relation should be considered.
- The order in which the comparisons of the significant fields are carried out can influence the performance of the equals comparison.

Example 6.6 Implications of overriding the `equals()` *Method*

```
public class TestUsableVNO {
  public static void main(String[] args) {
    // Three individual version numbers.
    UsableVNO latest  = new UsableVNO(9,1,1);                          // (1)
    UsableVNO inShops = new UsableVNO(9,1,1);                          // (2)
    UsableVNO older   = 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)};

    // An array with number of downloads.
    Integer[] downloads = {245, 786, 54,1010, 123};                    // (5)

    TestCaseVNO.test(latest, inShops, older, versions, downloads);     // (6)
  }
}
```

- Possible output from the program:

```
class UsableVNO
Test object reference and value equality:
```

```
    latest: (9.1.1), inShops: (9.1.1), older: (6.6.6)
    latest == inShops:      false
    latest.equals(inShops): false
    latest == older:        false
    latest.equals(older):   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: false
List: [(3.49.1), (8.19.81), (2.48.28), (10.23.78), (9.1.1)]
    Search key (9.1.1) contained in list: false
Map: {(2.48.28)=54, (9.1.1)=123, (3.49.1)=245, (8.19.81)=786, (10.23.78)=1010}
    Hash code for keys in the map:
      (3.49.1): 31168322
     (8.19.81): 17225372
     (2.48.28): 5433634
    (10.23.78): 2430287
       (9.1.1): 17689166
    Search key (9.1.1) has hash code: 6585861
    Map contains search key (9.1.1): false
Exception in thread "main" java.lang.ClassCastException: SimpleVNO cannot be cast
to java.lang.Comparable
      ...
      at TestCaseVNO.test(TestCaseVNO.java:66)
```

```
        at TestSimpleVNO.main(TestSimpleVNO.java:18)
```

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 using a *hash function*.
  - The array index returned by the hash function is called the *hash code* of the item—also called the *hash value* of the item.
  - The hash code identifies a particular bucket.
- Storing an item involves the following steps:
  1. Hashing the item to determine the bucket.
  2. If the item does not match one already in the bucket, it is stored in the bucket.
  - Note that no duplicate items are stored.
- Retrieving an item is based on using a *key*.
  1. Hashing the key to determine the bucket.
  2. If the key matches an item in the bucket, this item is retrieved from the bucket.
- *Managing Collisions*: Different items can hash to the same bucket.
  - A hash-based storage scheme needs a hash function and an equality function.

- a `hashCode()` method that produces hash codes for the objects
- an `equals()` method that tests objects for equality

- *Hash Code Distribution*: Should produce a uniform distribution of hash codes for a collection of items across all possible hash codes.
- *Hash table:* Contains *key-value entries*.
- As a general rule for implementing these methods, *a class that overrides the* `equals()` *method must override the* `hashCode()` *method*.

General Contract of the `hashCode()` Method

- The general contract of the `hashCode()` method stipulates:
  - *Consistency during execution.*
  - *Object value equality implies hash code equality.*
  - *Object value inequality places no restrictions on the hash code*: It is strongly recommended that the `hashCode()` method produce unequal hash codes for unequal objects.
- Note that the hash contract does not imply that objects with equal hash codes are equal.

Heuristics for Implementing the `hashCode()` Method

- Each significant field is included in the computation.
  - Only the fields that have bearing on the `equals()` method are included.
  - Setup below ensures that the result from incorporating a field value is used to calculate the contribution from the next field value.

```
hashValue = 11 * 313 + release * 312 + revision * 311 + patch
```

Example 6.7 Implementing the `hashCode()` *Method*

```
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)
```

```
    if (obj == this)                               // (2)
      return true;
    if (!(obj instanceof ReliableVNO))             // (3)
      return false;
    ReliableVNO vno = (ReliableVNO) obj;           // (4)

    return Objects.equals(vno.patch, this.patch)    // (5)
        && Objects.equals(vno.revision, this.revision)
        && Objects.equals(vno.release, this.release);
  }

  @Override public int hashCode() {                // (6)
    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); // (6b)
  }
}
```

- 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.

```
public int hashCode() {       // Legal but insufficient
  return 1949;
}
```

- For immutable objects, the hash code can be cached, that is, calculated once and returned whenever the `hashCode()` method is called.
- The numeric wrapper types, the `Boolean`, and `String` classes provide the `hashCode()` method.

```
public class TestReliableVNO {
  public static void main(String[] args) {
    // Three individual version numbers.
    ReliableVNO latest  = new ReliableVNO(9,1,1);                          // (1)
    ReliableVNO inShops = new ReliableVNO(9,1,1);                          // (2)
    ReliableVNO older   = new ReliableVNO(6,6,6);                          // (3)

    // An array of version numbers.
    ReliableVNO[] versions =  new ReliableVNO[] {                          // (4)
        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)};

    // An array with number of downloads.
```

```
    Integer[] downloads = {245, 786, 54,1010, 123};                       // (5)

    TestCaseVNO.test(latest, inShops, older, versions, downloads);        // (6)
  }
}
```

- Possible output from the program:

```
class ReliableVNO
...
Map: {(10.23.78)=1010, (2.48.28)=54, (9.1.1)=123, (3.49.1)=245, (8.19.81)=786}
    Hash code for keys in the map:
      (3.49.1): 332104
     (8.19.81): 336059
     (2.48.28): 331139
    (10.23.78): 338102
       (9.1.1): 336382
    Search key (9.1.1) has hash code: 336382
    Map contains search key (9.1.1): true
Exception in thread "main" java.lang.ClassCastException: ReliableVNO cannot be
cast to java.lang.Comparable
...
```

Comparing Objects

- In order to sort the objects of a class, it should be possible to *compare* the objects.
- The criteria used to do the comparison depends on what is meaningful for the class.
- A *total ordering* allows objects to be compared.
- 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, i.e. treats upper case characters as lower case 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 generic `Comparable<E>` interface.
- Other total orderings can be defined by providing a *comparator* that implements the `Comparator<E>` interface.

The `Comparable<E>` Interface

- The generic `Comparable<E>` interface is implemented by the *objects of the class*.

> `int compareTo(E other)`
> It returns a negative integer, zero, or a positive integer if the current object is less than, equal to, or greater than the specified object, 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 `NullPointerException` if the argument is `null`.

- 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 manually using the `Collections.sort()` or `List.sort()` methods

Criteria for Implementing the `compareTo()` method

- 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, i.e., 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, if `obj1.compareTo(obj2) > 0` and `obj2.compareTo(obj3) > 0`, then `obj1.compareTo(obj3) > 0`.
- For any two objects of the class, which compare as equal, the `compareTo()` method must return the same result if these two objects are compared with any other object, i.e., the comparison is *congruent*.
  - i.e., the `compareTo()` method must be *consistent with equals*, that is, `(obj1.compareTo(obj2) == 0) == (obj1.equals(obj2))`. *This is recommended!*

```
@Override public boolean equals(Object other) {
  // ...
  return this.compareTo((Whatever)other) == 0;
}
```

- The magnitude of non-zero values returned by the `compareTo()` method is immaterial; the *sign* indicates the result of the comparison.
- An implementation of the `compareTo()` method for version numbers.

```
public final class VersionNumber implements Comparable<VersionNumber> {
```

```
    ...
    @Override public int compareTo(VersionNumber vno) {          // (7)
    ...
    }
    ...
}
```

- 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.

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

Example 6.8 Implementing the `compareTo()` *Method of the* `Comparable<E>` *Interface*

```
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)
```

```
    if (obj == this)                                  // (2)
      return true;
    if (!(obj instanceof VersionNumber))              // (3)
      return false;
    VersionNumber vno = (VersionNumber) obj;          // (4)
    return Objects.equals(vno.patch, this.patch)      // (5b)
        && Objects.equals(vno.revision, this.revision)
        && Objects.equals(vno.release, this.release);
  }

  @Override public int hashCode() {                             // (6)
    return Objects.hash(this.release, this.revision, this.patch); // (6b)
  }

  @Override public int compareTo(VersionNumber vno) { // (7)
    // Compare the release numbers.                      (8)
    if (this.release != vno.release)
      return Integer.compare(this.release, vno.release);

    // Release numbers are equal,                        (9)
    // must compare revision numbers.
    if (this.revision != vno.revision)
      return Integer.compare(this.revision, vno.revision);

    // Release and revision numbers are equal,           (10)
```

```
    // patch numbers determine the ordering.
    return Integer.compare(this.patch, vno.patch);
  }

  /*
  @Override public int compareTo(VersionNumber vno) {                // (11)
    return Comparator.comparingInt(VersionNumber::getRelease)        // (12)
                     .thenComparingInt(VersionNumber::getRevision)  // (13)
                     .thenComparingInt(VersionNumber::getPatch)     // (14)
                     .compare(this, vno);                            // (15)
  }
  */
}
```

- A `compareTo()` method is seldom implemented to interoperate with objects of other classes.

```
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;     // No such method in StringBuilder.
```

- The fields are compared with the most significant field first and the least significant field last.
- Comparison of integer values in fields can be optimized:

```
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 :

```
int releaseDiff = release - vno.release;   // Can give incorrect result.
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.
- The method implementation relies exclusively on the methods of the `Comparator<E>` interface. It essentially uses a *conditional comparator* that applies its constituent

comparators conditionally in the order in which it composed.

Example 6.9 Implications of Implementing the `compareTo()` *Method*

```
public class TestVersionNumber {
  public static void main(String[] args) {
    // Three individual version numbers.
    VersionNumber latest  = new VersionNumber(9,1,1);                    // (1)
    VersionNumber inShops = new VersionNumber(9,1,1);                    // (2)
    VersionNumber older   = new VersionNumber(6,6,6);                    // (3)

    // An array of version numbers.
    VersionNumber[] versions =  new VersionNumber[] {                    // (4)
        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)};

    // An array with number of downloads.
    Integer[] downloads = {245, 786, 54,1010, 123};                      // (5)

    TestCaseVNO.test(latest, inShops, older, versions, downloads);       // (6)
  }
}
```

- Testing:

```
out.println("Sorted set:\n    " + (new TreeSet<>(vnoList)));      // (22)
out.println("Sorted map:\n    " +
```

```
            (new TreeMap<>(versionStatistics)));                  // (23)
Sorted set:
    [(2.48.28), (3.49.1), (8.19.81), (9.1.1), (10.23.78)]
Sorted map:
    {(2.48.28)=54, (3.49.1)=245, (8.19.81)=786, (9.1.1)=123, (10.23.78)=1010}
```

- We can run generic algorithms on collections of version numbers:

```
out.println("List before sorting: " + vnoList);
Collections.sort(vnoList, null);                    // (24) Natural order.
out.println("List after sorting:  " + vnoList);
```

```
List before sorting: [(3.49.1), (8.19.81), (2.48.28), (10.23.78), (9.1.1)]
List after sorting:  [(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)`:

```
int resultIndex = Collections.binarySearch(vnoList, searchKey, null);// (25)
out.printf("Binary search in list found key %s at index: %d%n",
            searchKey, resultIndex);
```

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

The Comparator<E> Interface

- The java.util.Comparator<E> interface is a *functional interface.*
- It is designated as 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.
- Imposes a specific *total ordering* on the elements.

```
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, i.e., 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.

- The Collections and Arrays classes provide utility methods for sorting, which take a Comparator.
- *Rhyming ordering*: If we reverse the two strings, "report" and "court", the reversed string "troper" is lexicographically less than the reversed string "truoc".

Example 6.10 Natural Ordering and Total Orderings

```
import java.util.Collections;
import java.util.Comparator;
import java.util.Set;
import java.util.TreeSet;

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

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

```
>java ComparatorUsage court Stuart report Resort assort support transport distort
```

- Output from the program:

```
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]
```

- *Reverse natural ordering*: The method `Comparator.reversedOrder()` readily returns a comparator that imposes the reverse of the natural ordering.

Example 6.11 Using a Comparator for Version Numbers

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

import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;
import java.util.List;

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);
  }
}
```

- Program output:

```
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 `static` and `default` methods in the `Comparator<E>` interface

```
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)`.

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

These methods return a `null`-friendly comparator that considers `null` to be either less than non-`null` or greater than non-`null`, respectively.

Useful comparators for sorting or searching in collections and maps when `null`s are considered as actual values.

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

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

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

Returns a `Comparator<T>` that applies `func` to given keys and compares the results using `cmp`, equivalent to `(a, b) -> cmp.compare(func.apply(a), func.apply(b))`.

`static <T> Comparator<T>`
`           comparing`*PrimType*`(To`*PrimType*`Function<? super T> func)`

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

A *primType* is either `int`, `long` or `double`, and the corresponding *PrimType* is `Int`, `Long` or `Double`.

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

Returns a *conditional* comparator which is composed from `this Comparator` and the `cmp Comparator`.

If `this Comparator` determines that given keys are equal, then `cmp` is used to determine the order. Effectively first executes `this.compare(a, b)`, then `cmp.compare(a, b)` if necessary.

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

Returns a conditional comparator which is composed of `this` `Comparator` and the `cmp` `Comparator`.

If `this` `Comparator` determines that given keys are equal, then applies `func` to given keys and the results are compared using `cmp`.

Effectively first executes `this.compare(a, b)`, then `cmp.compare(func.apply(a), func.apply(b))` if necessary.

```
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 given keys are equal, then applies `func` to given keys and the results compared by natural ordering. Effectively first executes `this.compare(a, b)`, then `func.apply(a).compareTo(func.apply(b))` if necessary.

`default Comparator<T>`
        `thenComparing`*PrimType*`(To`*PrimType*`Function<? super T> func)`

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

A *primType* is either `int`, `long` or `double`, and the corresponding *PrimType* is `Int`, `Long` or `Double`.