

Chapter 11: Collections and Maps

Implementing the equals(), hashCode() and compareTo() methods

A Programmer's Guide to Java Certification (Second Edition)

Khalid A. Mughal and Rolf W. Rasmussen
Addison-Wesley, 2nd. Edition, 2003.
ISBN 0-201-72828-1

E-mail: pgjc2e@ii.uib.no

Web site: <http://www.ii.uib.no/~khalid/pgjc2e/>

Permission is given to use these notes in connection with the book.

Modified: 8/3/06

11.7 Implementing the equals(), hashCode() and compareTo() methods

- The majority of the non-final methods of the Object class provide general contracts for objects, which the classes overriding the methods should honor.
- Objects of a class that override the equals() method can be used as elements in a collection.
- If they override the hashCode() method, they can also be used as elements in a HashSet and as keys in a HashMap.
- Implementing the Comparable interface allows them to be used as elements in sorted collections and sorted maps.

Example: Version Number (VNO)

- A *version number* (VNO) for a software product comprises three pieces of information:
 - a release number
 - a revision number
 - a patch number
- Ranking version numbers chronologically:
 - The release number is most *significant*.
 - The revision number is less significant than the release number, and the patch number is the least significant.

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 -- the most discriminating equivalence relation.

Example 11.7 A Simple Class for Version Number

```
public class SimpleVNO {
    // Does not override equals() or hashCode().

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

Using the SimpleVNO class

- See class `TestVersionSimple` in Example 11.8 which uses objects of the class `SimpleVNO`.
- Method `makeVersion()` creates a version number.
- The `versions` array and the `downloads` array represent a list of version numbers and their corresponding downloads, respectively.
- *Test object reference equality and object value equality.*
 - The result is always `false`, (6) - (9).
 - The `equals()` method is not overridden.
- *Test searching in an array.*
 - Search in the `versions` array will always fail, (10) - (13), as the `equals()` method is not overridden.
- *Test for membership in a collection.*
 - The result is always `false`, (14) - (15), as the `equals()` method is not overridden.
- *Test for membership in a map.*
 - The result is always `false`, (16) - (21), as the `hashCode()` method is not overridden.
- *Test for usage in a sorted collection and map.*
 - Exception thrown, (22) - (23), as the `compareTo()` method of the `Comparable` interface is not implemented.

Example 11.8 Implications of not overriding the `equals()` method

```
import java.util.*;

public class TestVersionSimple {
    public static void main(String[] args) {
        (new TestVersionSimple()).test();
    }

    protected Object makeVersion(int a, int b, int c) {
        return new SimpleVNO(a, b, c);
    }

    protected void test() {
        // Three individual version numbers.
        Object latest = makeVersion(9,1,1);           // (1)
        Object inShops = makeVersion(9,1,1);         // (2)
        Object older = makeVersion(6,6,6);           // (3)

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

        // An array of downloads.
```

```

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

// Various tests.
System.out.println("Test object reference and value equality:");
System.out.println("    latest: " + latest + ", inShops: " + inShops +
    ", older: " + older);
System.out.println("    latest == inShops: "      + (latest == inShops)); // (6)
System.out.println("    latest.equals(inShops): " + (latest.equals(inShops))); // (7)
System.out.println("    latest == older: "       + (latest == older)); // (8)
System.out.println("    latest.equals(older): "  + (latest.equals(older))); // (9)

Object searchKey = inShops;
System.out.println("Search key: " + searchKey); // (10)

System.out.print("Array: ");
for (int i = 0; i < versions.length; i++) // (11)
    System.out.print(versions[i] + " ");
boolean found = false;
for (int i = 0; i < versions.length && !found; i++)
    found = searchKey.equals(versions[i]); // (12)
System.out.println("\n    Search key found in array: " + found); // (13)

List vnoList = Arrays.asList(versions); // (14)
System.out.println("List: " + vnoList);

```

```

System.out.println("    Search key contained in list: " +
    vnoList.contains(searchKey)); // (15)

Map versionStatistics = new HashMap(); // (16)
for (int i = 0; i < versions.length; i++) // (17)
    versionStatistics.put(versions[i], downloads[i]);
System.out.println("Map: " + versionStatistics); // (18)
System.out.println("    Hash code for keys in the map:");
for (int i = 0; i < versions.length; i++) // (19)
    System.out.println("        " + versions[i] + ": " + versions[i].hashCode());
System.out.println("    Search key " + searchKey +
    " has hash code: " + searchKey.hashCode()); // (20)
System.out.println("    Map contains search key: " +
    versionStatistics.containsKey(searchKey)); // (21)

System.out.println("Sorted list:\n\t" + (new TreeSet(vnoList))); // (22)
System.out.println("Sorted map:\n\t" + (new TreeMap(versionStatistics))); // (23)

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

System.out.println("Binary search in list:"); // (25)
int resultIndex = Collections.binarySearch(vnoList, searchKey);
System.out.println("\tKey: " + searchKey + "\tKey index: " + resultIndex);
}
}

```

Output from the program:

```
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
Search key: (9.1.1)
Array: (3.49.1) (8.19.81) (2.48.28) (10.23.78) (9.1.1)
  Search key found in array: false
List: [(3.49.1), (8.19.81), (2.48.28), (10.23.78), (9.1.1)]
  Search key contained in list: false
Map: {(9.1.1)=123, (10.23.78)=1010, (8.19.81)=786, (3.49.1)=245, (2.48.28)=54}
  Hash code for keys in the map:
    (3.49.1): 13288040
    (8.19.81): 27355241
    (2.48.28): 30269696
    (10.23.78): 24052850
    (9.1.1): 26022015
  Search key (9.1.1) has hash code: 20392474
  Map contains search key: false
Exception in thread "main" java.lang.ClassCastException
...
```

Implementing equals() method

- *Equivalence relation:*
 - *Reflexive:* For any reference self, self.equals(self) is always true.
 - *Symmetric:* For any references x and y, x.equals(y) is true if and only if 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) always return the same result, provided the objects denoted by these references have not been modified to affect the equals comparison.
 - *null comparison:* For any non-null reference obj, obj.equals(null) is always false.
- The general contract of the equals() method is defined between *objects of arbitrary classes*.

Example 11.9 *Implementing the equals() method*

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

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

    public boolean equals(Object obj) {
        if (obj == this) // (1)
            return true; // (2)
        if (!(obj instanceof UsableVNO)) // (3)
            return false;
        UsableVNO vno = (UsableVNO) obj; // (4)
```

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

Checklist for implementing the equals() method

- See class UsableVNO in Example 11.9.

Method overriding signature

- The method prototype is:

```
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:

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

- 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 denoted by the argument.

```
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

- This is usually 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

- The equals() method in Example 11.9 checks the type of the argument object at (3), using the instanceof operator:

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

– The code above also does the null comparison correctly, returning false if the argument obj has the value null.

- The test at (3) can also be replaced by the following code in order to exclude all other objects, including subclass objects:

```
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.
- For fields that are of primitive data types, their primitive values can be compared.

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

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

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

```
(vno.productInfo == this.productInfo ||
 this.productInfo != null && this.productInfo.equals(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()).
- Only fields that have significance for the equivalence relation should be considered.
- The order in which the comparisons are carried out can influence the performance of the equals comparison.

Using the UsableVNO class

- See class TestVersionUsable in Example 11.10 which uses objects of the class UsableVNO.
- *Test object value equality.*
 - Object value equality is compared correctly as defined by the equals() method.
- *Test searching in an array.*
 - Search in the versions array works correctly according to the overridden equals() method.
- *Test for membership in a collection.*
 - Membership in the versions array works correctly according to the overridden equals() method.
- *Test for membership in a map.*
 - The result is always false as the hashCode() method is not overridden.
- *Test for usage in a sorted collection and map.*
 - Exception thrown as the compareTo() method of the Comparable interface is not implemented.

Example 11.10 *Implications of overriding the equals() method*

```
public class TestVersionUsable extends TestVersionSimple {
    public static void main(String[] args) {
        (new TestVersionUsable()).test();
    }
    protected Object makeVersion(int a, int b, int c) {
        return new UsableVNO(a, b, c);
    }
}
```

Output from the program:

```
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
Search key: (9.1.1)
Array: (3.49.1) (8.19.81) (2.48.28) (10.23.78) (9.1.1)
  Search key found in array: true
List: [(3.49.1), (8.19.81), (2.48.28), (10.23.78), (9.1.1)]
  Search key contained in list: true
Map: {(10.23.78)=1010, (2.48.28)=54, (3.49.1)=245, (9.1.1)=123, (8.19.81)=786}
Hash code for keys in the map:
```

```
(3.49.1): 27355241
(8.19.81): 30269696
(2.48.28): 24052850
(10.23.78): 26022015
(9.1.1): 3541984
Search key (9.1.1) has hash code: 11352996
Map contains search key: false
Exception in thread "main" java.lang.ClassCastException
...
```

Hashing

- *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*.
 - 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 value* of the item.
- 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*. The key represents the identify the item. Item retrieval is also a two-step process:
 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.
- Different items can hash to the same bucket, meaning that the hash function returns the same hash value for these items: *collisions*.
 - Finding an item in the bucket can entail a search, and requires an equality function to compare items.

- An ideal hash function produces a uniform distribution of hash values for a collection of items across all possible hash values.
- Not an easy task to design adequate hash functions, but fortunately, there are heuristics.

The hashCode() method

- A *hash table* contains *key-value entries* as items, and the hashing is done only on the keys to provide efficient lookup of values.
 - Matching a given key with a key in an entry, determines the value.
- To use hash-based collections and maps of the `java.util` package, the class must provide appropriate implementations of the following methods from the `Object` class:
 - A `hashCode()` method that produces hash values for the objects.
 - An `equals()` method that tests objects for equality.
- General rule: *a class which overrides the `equals()` method must override the `hashCode()` method.*
- The class `UsableVNO` violates the key tenet of the `hashCode()` contract: *equal objects must produce equal hash codes.*

General Contract of the hashCode() Method

- *Consistency during execution.*
- *Object value equality implies hash value equality.*
- *Object value inequality places no restrictions on the hash value.*

Note that the hash 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 will hash to the same bucket.

Example 11.11 *Implementing the hashCode() method*

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

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

    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 vno.patch == this.patch &&           // (5)
            vno.revision == this.revision &&
            vno.release == this.release;
    }

    public int hashCode() {                            // (6)
        int hashValue = 11;
        hashValue = 31 * hashValue + release;
        hashValue = 31 * hashValue + revision;
        hashValue = 31 * hashValue + patch;
        return hashValue;
    }
}
```

Heuristics for implementing the hashCode() Method

- In Example 11.11, the hash value is computed according to the following formula:

$\text{hashValue} = 11 * 31^3 + \text{release} * 31^2 + \text{revision} * 31^1 + \text{patch}$

- Each significant field is included in the computation.
- This ensures that objects which are equal according to the equals() method, also have equal hash values according to the hashCode() method.

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

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

- Calculating the hash value sfVal for a significant field sf depends on the type of the field:

- Field sf is boolean: $\text{sfVal} = \text{sf} ? 0 : 1$
- Field sf is byte, char, short, or int: $\text{sfVal} = (\text{int})\text{sf}$
- Field sf is long: $\text{sfVal} = (\text{int}) (\text{sf} \wedge (\text{sf} \ggg 32))$
- Field sf is float: $\text{sfVal} = \text{Float.floatToInt}(\text{sf})$
- Field sf is double: $\text{long sfValTemp} = \text{Double.doubleToLong}(\text{sf});$
 $\text{sfVal} = (\text{int}) (\text{sfValTemp} \wedge (\text{sfValTemp} \ggg 32))$
- Field sf is a reference that denotes an object. Typically, the hashCode() method is invoked recursively if the equals() method is invoked recursively:
 $\text{sfVal} = (\text{sf} == \text{null} ? 0 : \text{sf.hashCode}())$
- Field sf is an array. 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 value.
- A legal or correct hash function does not necessarily mean it is appropriate or efficient.

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

Example 11.12 *Implications of overriding the hashCode() method*

```
public class TestVersionReliable extends TestVersionSimple {
    public static void main(String[] args) {
        (new TestVersionReliable()).test();
    }
    protected Object makeVersion(int a, int b, int c) {
        return new ReliableVNO(a, b, c);
    }
}
```

Output from the program:

```
...
Map: {(10.23.78)=1010, (2.48.28)=54, (3.49.1)=245, (8.19.81)=786, (9.1.1)=123}
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: true
Exception in thread "main" java.lang.ClassCastException
...
```

The compareTo() method

- This method defines the *natural ordering* for the instances of a class.
- Objects implementing Comparable can be used in sorted collections and sorted maps.

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

The method 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 order. It throws a `ClassCastException` if the reference value passed in the argument cannot be cast to the type of the current object.

- 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.
 - All three order comparison relations (*less than*, *equal to*, *greater than*) embodied in the `compareTo()` method must be *transitive*.
 - For any two objects of the class, which compare as equal, the `compareTo()` method must return the same results if these two objects are compared with any other object.
 - The `compareTo()` method must be *consistent with equals*.
- The magnitude of non-zero values returned by the method is immaterial; the sign indicates the result of the comparison.

Example 11.13 *Implementing the `compareTo()` method of the `Comparable` interface*

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

    public boolean equals(Object obj) { // (1)
        if (this == obj) // (2)
            return true;
        if (!(obj instanceof VersionNumber)) // (3)
            return false;
        VersionNumber vno = (VersionNumber) obj; // (4)
        return vno.patch == this.patch && // (5)
            vno.revision == this.revision &&
            vno.release == this.release;
    }

    public int hashCode() { // (6)
        int hashValue = 11;
        hashValue = 31 * hashValue + release;
        hashValue = 31 * hashValue + revision;
        hashValue = 31 * hashValue + patch;
        return hashValue;
    }
}
```

```

public int compareTo(VersionNumber vno2) {           // (7)
    if (this == vno2)                               // (8)
        return 0;
    // Compare the release numbers.                 (9)
    if (release < vno2.release)
        return -1;
    if (release > vno2.release)
        return 1;

    // Release numbers are equal,                    (10)
    // must compare revision numbers.
    if (revision < vno2.revision)
        return -1;
    if (revision > vno2.revision)
        return 1;

    // Release and revision numbers are equal,      (11)
    // must compare patch numbers.
    if (patch < vno2.patch)
        return -1;
    if (patch > vno2.patch)
        return 1;

    // All fields are equal.                          (12)
    return 0;
}

```

```

}

```

- A null argument should result in a `NullPointerException`.
- The fields are compared with the most significant field first and the least significant field last.
- 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 invoke the `compareTo()` method on them, or to use a comparator.
- The `equals()` implementation can call the `compareTo()` method.

```

public boolean equals(Object other) {
    // ...
    return compareTo((VersionNumber)other) == 0; // Note the cast.
}

```

Example 11.14 *Implications of implementing the compareTo() method*

```
public class TestVersion extends TestVersionSimple {
    public static void main(String[] args) {
        (new TestVersion()).test();
    }
    protected Object makeVersion(int a, int b, int c) {
        return new VersionNumber(a, b, c);
    }
}
```

Output from the program:

```
...
Sorted list:
  [(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}
...
```