The Java™ Tutorials

Trail: Collections
Lesson: Interfaces

The Java Tutorials have been written for JDK 8. Examples and practices described in this page don't take advantage of improvements introduced in later releases.

Object Ordering

A `List l` may be sorted as follows.

```
Collections.sort(l);
```

If the `List` consists of `String` elements, it will be sorted into alphabetical order. If it consists of `Date` elements, it will be sorted into chronological order. How does this happen? `String` and `Date` both implement the `Comparable` interface. `Comparable` implementations provide a *natural ordering* for a class, which allows objects of that class to be sorted automatically. The following table summarizes some of the more important Java platform classes that implement `Comparable`.

Classes Implementing Comparable

Class	Natural Ordering
`Byte`	Signed numerical
`Character`	Unsigned numerical
`Long`	Signed numerical
`Integer`	Signed numerical
`Short`	Signed numerical
`Double`	Signed numerical
`Float`	Signed numerical
`BigInteger`	Signed numerical
`BigDecimal`	Signed numerical

`Boolean`	`Boolean.FALSE < Boolean.TRUE`
`File`	System-dependent lexicographic on path name
`String`	Lexicographic
`Date`	Chronological
`CollationKey`	Locale-specific lexicographic

If you try to sort a list, the elements of which do not implement `Comparable`, `Collections.sort(list)` will throw a `ClassCastException`. Similarly, `Collections.sort(list, comparator)` will throw a `ClassCastException` if you try to sort a list whose elements cannot be compared to one another using the `comparator`. Elements that can be compared to one another are called *mutually comparable*. Although elements of different types may be mutually comparable, none of the classes listed here permit interclass comparison.

This is all you really need to know about the `Comparable` interface if you just want to sort lists of comparable elements or to create sorted collections of them. The next section will be of interest to you if you want to implement your own `Comparable` type.

Writing Your Own Comparable Types

The `Comparable` interface consists of the following method.

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

The `compareTo` method compares the receiving object with the specified object and returns a negative integer, 0, or a positive integer depending on whether the receiving object is less than, equal to, or greater than the specified object. If the specified object cannot be compared to the receiving object, the method throws a `ClassCastException`.

The `following class representing a person's name` implements `Comparable`.

```
import java.util.*;

public class Name implements Comparable<Name> {
    private final String firstName, lastName;

    public Name(String firstName, String lastName) {
        if (firstName == null || lastName == null)
            throw new NullPointerException();
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public String firstName() { return firstName; }
```

```
    public String lastName()  { return lastName;  }

    public boolean equals(Object o) {
        if (!(o instanceof Name))
            return false;
        Name n = (Name) o;
        return n.firstName.equals(firstName) && n.lastName.equals(lastName);
    }

    public int hashCode() {
        return 31*firstName.hashCode() + lastName.hashCode();
    }

    public String toString() {
        return firstName + " " + lastName;
    }

    public int compareTo(Name n) {
        int lastCmp = lastName.compareTo(n.lastName);
        return (lastCmp != 0 ? lastCmp : firstName.compareTo(n.firstName));
    }
}
```

To keep the preceding example short, the class is somewhat limited: It doesn't support middle names, it demands both a first and a last name, and it is not internationalized in any way. Nonetheless, it illustrates the following important points:

- `Name` objects are *immutable*. All other things being equal, immutable types are the way to go, especially for objects that will be used as elements in `Set`s or as keys in `Map`s. These collections will break if you modify their elements or keys while they're in the collection.
- The constructor checks its arguments for `null`. This ensures that all `Name` objects are well formed so that none of the other methods will ever throw a `NullPointerException`.
- The `hashCode` method is redefined. This is essential for any class that redefines the `equals` method. (Equal objects must have equal hash codes.)
- The `equals` method returns `false` if the specified object is `null` or of an inappropriate type. The `compareTo` method throws a runtime exception under these circumstances. Both of these behaviors are required by the general contracts of the respective methods.
- The `toString` method has been redefined so it prints the `Name` in human-readable form. This is always a good idea, especially for objects that are going to get put into collections. The various collection types' `toString` methods depend on the `toString` methods of their elements, keys, and values.

Since this section is about element ordering, let's talk a bit more about `Name`'s `compareTo` method. It implements the standard name-ordering algorithm, where last names take precedence over first names. This is exactly what you want in a natural ordering. It would be very confusing indeed if the natural ordering were unnatural!

Take a look at how `compareTo` is implemented, because it's quite typical. First, you compare the most significant part of the object (in this case, the last name). Often, you can just use the natural ordering of the part's type. In this case, the part is a `String` and the natural (lexicographic) ordering is exactly what's called for. If the comparison results in anything other than zero, which represents equality, you're done: You just

return the result. If the most significant parts are equal, you go on to compare the next most-significant parts. In this case, there are only two parts — first name and last name. If there were more parts, you'd proceed in the obvious fashion, comparing parts until you found two that weren't equal or you were comparing the least-significant parts, at which point you'd return the result of the comparison.

Just to show that it all works, here's `a program that builds a list of names and sorts them`.

```
import java.util.*;

public class NameSort {
    public static void main(String[] args) {
        Name nameArray[] = {
            new Name("John", "Smith"),
            new Name("Karl", "Ng"),
            new Name("Jeff", "Smith"),
            new Name("Tom", "Rich")
        };

        List<Name> names = Arrays.asList(nameArray);
        Collections.sort(names);
        System.out.println(names);
    }
}
```

If you run this program, here's what it prints.

```
[Karl Ng, Tom Rich, Jeff Smith, John Smith]
```

There are four restrictions on the behavior of the `compareTo` method, which we won't go into now because they're fairly technical and boring and are better left in the API documentation. It's really important that all classes that implement `Comparable` obey these restrictions, so read the documentation for `Comparable` if you're writing a class that implements it. Attempting to sort a list of objects that violate the restrictions has undefined behavior. Technically speaking, these restrictions ensure that the natural ordering is a *total order* on the objects of a class that implements it; this is necessary to ensure that sorting is well defined.

Comparators

What if you want to sort some objects in an order other than their natural ordering? Or what if you want to sort some objects that don't implement `Comparable`? To do either of these things, you'll need to provide a `Comparator` — an object that encapsulates an ordering. Like the `Comparable` interface, the `Comparator` interface consists of a single method.

```
public interface Comparator<T> {
    int compare(T o1, T o2);
}
```

The `compare` method compares its two arguments, returning a negative integer, 0, or a positive integer depending on whether the first argument is less than, equal to, or greater than the second. If either of the arguments has an inappropriate type for the `Comparator`, the `compare` method throws a `ClassCastException`.

Much of what was said about `Comparable` applies to `Comparator` as well. Writing a `compare` method is nearly identical to writing a `compareTo` method, except that the former gets both objects passed in as arguments. The `compare` method has to obey the same four technical restrictions as `Comparable`'s `compareTo` method for the same reason — a `Comparator` must induce a total order on the objects it compares.

Suppose you have a class called `Employee`, as follows.

```
public class Employee implements Comparable<Employee> {
    public Name name()     { ... }
    public int number()    { ... }
    public Date hireDate() { ... }
       ...
}
```

Let's assume that the natural ordering of `Employee` instances is `Name` ordering (as defined in the previous example) on employee name. Unfortunately, the boss has asked for a list of employees in order of seniority. This means we have to do some work, but not much. The following program will produce the required list.

```
import java.util.*;
public class EmpSort {
    static final Comparator<Employee> SENIORITY_ORDER =
                                      new Comparator<Employee>() {
            public int compare(Employee e1, Employee e2) {
                return e2.hireDate().compareTo(e1.hireDate());
            }
    };

    // Employee database
    static final Collection<Employee> employees = ... ;

    public static void main(String[] args) {
        List<Employee> e = new ArrayList<Employee>(employees);
        Collections.sort(e, SENIORITY_ORDER);
        System.out.println(e);
    }
}
```

The `Comparator` in the program is reasonably straightforward. It relies on the natural ordering of `Date` applied to the values returned by the `hireDate` accessor method. Note that the `Comparator` passes the hire date of its second argument to its first rather than vice versa. The reason is that the employee who was hired most recently is the least senior; sorting in the order of hire date would put the list in reverse seniority order. Another technique people sometimes use to achieve this effect is to maintain the argument order but to negate the result of the comparison.

```
// Don't do this!!
return -r1.hireDate().compareTo(r2.hireDate());
```

You should always use the former technique in favor of the latter because the latter is not guaranteed to work. The reason for this is that the `compareTo` method can return any negative `int` if its argument is less than the object on which it is invoked. There is one negative `int` that remains negative when negated, strange as it may seem.

```
-Integer.MIN_VALUE == Integer.MIN_VALUE
```

The `Comparator` in the preceding program works fine for sorting a `List`, but it does have one deficiency: It cannot be used to order a sorted collection, such as `TreeSet`, because it generates an ordering that is *not compatible with* equals. This means that this `Comparator` equates objects that the `equals` method does not. In particular, any two employees who were hired on the same date will compare as equal. When you're sorting a `List`, this doesn't matter; but when you're using the `Comparator` to order a sorted collection, it's fatal. If you use this `Comparator` to insert multiple employees hired on the same date into a `TreeSet`, only the first one will be added to the set; the second will be seen as a duplicate element and will be ignored.

To fix this problem, simply tweak the `Comparator` so that it produces an ordering that *is compatible with* `equals`. In other words, tweak it so that the only elements seen as equal when using `compare` are those that are also seen as equal when compared using `equals`. The way to do this is to perform a two-part comparison (as for `Name`), where the first part is the one we're interested in — in this case, the hire date — and the second part is an attribute that uniquely identifies the object. Here the employee number is the obvious attribute. This is the `Comparator` that results.

```
static final Comparator<Employee> SENIORITY_ORDER =
                                        new Comparator<Employee>() {
    public int compare(Employee e1, Employee e2) {
        int dateCmp = e2.hireDate().compareTo(e1.hireDate());
        if (dateCmp != 0)
            return dateCmp;

        return (e1.number() < e2.number() ? -1 :
                (e1.number() == e2.number() ? 0 : 1));
    }
};
```

One last note: You might be tempted to replace the final `return` statement in the `Comparator` with the simpler:

```
return e1.number() - e2.number();
```

Don't do it unless you're *absolutely sure* no one will ever have a negative employee number! This trick does not work in general because the signed integer type is not big enough to represent the difference of two arbitrary signed integers. If `i` is a large positive integer and `j` is a large negative integer, `i - j` will overflow and will return a negative integer. The resulting `comparator` violates one of the four technical restrictions we keep talking about (transitivity) and produces horrible, subtle bugs. This is not a purely theoretical concern; people get burned by it.

Previous page: The Map Interface
Next page: The SortedSet Interface