Ordering

Example

```
assertTrue(byLengthOrdering.reverse().isOrdered(list));
```

Overview

`Ordering` is Guava's "fluent" `Comparator` class, which can be used to build complex comparators and apply them to collections of objects.

At its core, an `Ordering` instance is nothing more than a special `Comparator` instance. `Ordering` simply takes the methods that rely on a `Comparator` (for example, `Collections.max`) and makes them available as instance methods. For additional power, `Ordering` class provides chaining methods to tweak and enhance existing comparators.

Creation

Common orderings are provided by static methods:

Method	Description
`natural()`	Uses the *natural ordering* on Comparable types.
`usingToString()`	Compares objects by the lexicographical ordering of their string representations, as returned by `toString()`.

Making a preexisting `Comparator` into an `Ordering` is as simple as using `Ordering.from(Comparator)`.

But the more common way to create a custom `Ordering` is to skip the `Comparator` entirely in favor of extending the `Ordering` abstract class directly:

```
Ordering<String> byLengthOrdering = new Ordering<String>() {
  public int compare(String left, String right) {
    return Ints.compare(left.length(), right.length());
  }
};
```

Chaining

A given `Ordering` can be wrapped to obtain derived orderings. Some of the most commonly used variations include:

Method	Description
`reverse()`	Returns the reverse ordering.
`nullsFirst()`	Returns an `Ordering` that orders nulls before non-null elements, and otherwise behaves the same as the original `Ordering`. See also `nullsLast()`
`compound(Comparator)`	Returns an `Ordering` which uses the specified `Comparator` to "break ties."
`lexicographical()`	Returns an `Ordering` that orders iterables lexicographically by their elements.
`onResultOf(Function)`	Returns an `Ordering` which orders values by applying the function to them and then comparing the results using the original `Ordering`.

For example, let's say you want a comparator for the class...

```
class Foo {
  @Nullable String sortedBy;
  int notSortedBy;
}
```

...that can deal with null values of `sortedBy`. Here is a solution built atop the chaining methods:

```
Ordering<Foo> ordering = Ordering.natural().nullsFirst().onResultOf(new Function<Foo, String
  public String apply(Foo foo) {
    return foo.sortedBy;
  }
});
```

When reading a chain of `Ordering` calls, work "backward" from right to left. The example above orders `Foo` instances by looking up their `sortedBy` field values, first moving any null `sortedBy` values to the top and then sorting the remaining values by natural string ordering. This backward order arises because

each chaining call is "wrapping" the previous `Ordering` into a new one.

(Exception to the "backwards" rule: For chains of calls to `compound`, read from left to right. To avoid confusion, avoid intermixing `compound` calls with other chained calls.)

Chains longer than a few calls can be difficult to understand. We recommend limiting chaining to about three calls as in the example above. Even then, you may wish to simplify the code by separating out intermediate objects such as `Function` instances:

```
Ordering<Foo> ordering = Ordering.natural().nullsFirst().onResultOf(sortKeyFunction);
```

Application

Guava provides a number of methods to manipulate or examine values or collections using the ordering. We list some of the most popular here.

Method	Description	See also
`greatestOf(Iterable iterable, int k)`	Returns the `k` greatest elements of the specified iterable, according to this ordering, in order from greatest to least. Not necessarily stable.	`leastOf`
`isOrdered(Iterable)`	Tests if the specified `Iterable` is in nondecreasing order according to this ordering.	`isStrictlyOrdered`
`sortedCopy(Iterable)`	Returns a sorted copy of the specified elements as a `List`.	`immutableSortedCopy`
`min(E, E)`	Returns the minimum of its two arguments according to this ordering. If the values compare as equal, the first argument is returned.	`max(E, E)`
`min(E, E, E, E...)`	Returns the minimum of its arguments according to this ordering. If there are multiple least values, the first is returned.	`max(E, E, E, E...)`
`min(Iterable)`	Returns the minimum element of the specified `Iterable`. Throws a `NoSuchElementException` if the `Iterable` is empty.	`max(Iterable)`, `min(Iterator)`, `max(Iterator)`