

ZoneOffsetTransitionRule.TimeDefinition

public interface **Comparable**<T>

This interface imposes a total ordering on the objects of each class that implements it. This ordering is referred to as the class's *natural ordering*, and the class's `compareTo` method is referred to as its *natural comparison method*.

Lists (and arrays) of objects that implement this interface can be sorted automatically by `Collections.sort` (and `Arrays.sort`). Objects that implement this interface can be used as keys in a *sorted map* or as elements in a *sorted set*, without the need to specify a *comparator*.

The natural ordering for a class *C* is said to be *consistent with equals* if and only if `e1.compareTo(e2) == 0` has the same boolean value as `e1.equals(e2)` for every *e1* and *e2* of class *C*. Note that `null` is not an instance of any class, and `e.compareTo(null)` should throw a `NullPointerException` even though `e.equals(null)` returns `false`.

It is strongly recommended (though not required) that natural orderings be consistent with `equals`. This is so because sorted sets (and sorted maps) without explicit comparators behave "strangely" when they are used with elements (or keys) whose natural ordering is inconsistent with `equals`. In particular, such a sorted set (or sorted map) violates the general contract for set (or map), which is defined in terms of the `equals` method.

For example, if one adds two keys *a* and *b* such that `(!a.equals(b) && a.compareTo(b) == 0)` to a sorted set that does not use an explicit comparator, the second add operation returns `false` (and the size of the sorted set does not increase) because *a* and *b* are equivalent from the sorted set's perspective.

Virtually all Java core classes that implement `Comparable` have natural orderings that are consistent with `equals`. One exception is `java.math.BigDecimal`, whose natural ordering equates `BigDecimal` objects with equal values and different precisions (such as 4.0 and 4.00).

For the mathematically inclined, the *relation* that defines the natural ordering on a given class *C* is:

$$\{(x, y) \text{ such that } x.compareTo(y) \leq 0\}.$$

The *quotient* for this total order is:

$$\{(x, y) \text{ such that } x.compareTo(y) == 0\}.$$

It follows immediately from the contract for `compareTo` that the quotient is an *equivalence relation* on *C*, and that the natural ordering is a *total order* on *C*. When we say that a class's natural ordering is *consistent with equals*, we mean that the quotient for the natural ordering is the equivalence relation defined by the class's `equals(Object)` method:

$$\{(x, y) \text{ such that } x.equals(y)\}.$$

This interface is a member of the [Java Collections Framework](#).

Since:

1.2

See Also:[Comparator](#)**Method Summary****All Methods Instance Methods Abstract Methods**

Modifier and Type	Method and Description
int	<code>compareTo(T o)</code> Compares this object with the specified object for order.

Method Detail**`compareTo`**

```
int compareTo(T o)
```

Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

The implementor must ensure `sgn(x.compareTo(y)) == -sgn(y.compareTo(x))` for all `x` and `y`. (This implies that `x.compareTo(y)` must throw an exception iff `y.compareTo(x)` throws an exception.)

The implementor must also ensure that the relation is transitive: `(x.compareTo(y)>0 && y.compareTo(z)>0)` implies `x.compareTo(z)>0`.

Finally, the implementor must ensure that `x.compareTo(y)==0` implies that `sgn(x.compareTo(z)) == sgn(y.compareTo(z))`, for all `z`.

It is strongly recommended, but *not* strictly required that `(x.compareTo(y)==0) == (x.equals(y))`. Generally speaking, any class that implements the `Comparable` interface and violates this condition should clearly indicate this fact. The recommended language is "Note: this class has a natural ordering that is inconsistent with equals."

In the foregoing description, the notation `sgn(expression)` designates the mathematical *signum* function, which is defined to return one of `-1`, `0`, or `1` according to whether the value of *expression* is negative, zero or positive.

Parameters:

`o` - the object to be compared.

Returns:

a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

Throws:

`NullPointerException` - if the specified object is null