Ranges

Example

```
List<Double> scores;
Iterable<Double> belowMedianScores = Iterables.filter(scores,
Range.lessThan(median));
...
Range<Integer> validGrades = Range.closed(1, 12);
for(int grade : ContiguousSet.create(validGrades, DiscreteDomain.integers())) {
  ...
}
```

Introduction

A range, sometimes known as an interval, is a convex (informally, "contiguous" or "unbroken") portion of a particular domain. Formally, convexity means that for any `a <= b <= c`, `range.contains(a) && range.contains(c)` implies that `range.contains(b)`.

Ranges may "extend to infinity" -- for example, the range `x > 3` contains arbitrarily large values -- or may be finitely constrained, for example `2 <= x < 5`. We will use the more compact notation, familiar to programmers with a math background:

- `(a..b) = {x | a < x < b}`
- `[a..b] = {x | a <= x <= b}`
- `[a..b) = {x | a <= x < b}`
- `(a..b] = {x | a < x <= b}`
- `(a..+∞) = {x | x > a}`
- `[a..+∞) = {x | x >= a}`
- `(-∞..b) = {x | x < b}`
- `(-∞..b] = {x | x <= b}`
- `(-∞..+∞) = all values`

The values a and b used above are called endpoints. To improve consistency, Guava's notion of `Range` requires that the upper endpoint may not be less than the lower endpoint. The endpoints may be equal only if at least one of the bounds is closed:

- `[a..a]` : singleton range
- `[a..a); (a..a]` : empty, but valid
- `(a..a)` : invalid

A range in Guava has the type `Range<C>`. All ranges are *immutable*.

Building Ranges

Ranges can be obtained from the static methods on `Range`:

Range type	Method

(a..b)	open(C, C)
[a..b]	closed(C, C)
[a..b)	closedOpen(C, C)
(a..b]	openClosed(C, C)
(a..+∞)	greaterThan(C)
[a..+∞)	atLeast(C)
(-∞..b)	lessThan(C)
(-∞..b]	atMost(C)
(-∞..+∞)	all()

```
Range.closed("left", "right"); // all strings lexographically between "left" and
"right" inclusive
Range.lessThan(4.0); // double values strictly less than 4
```

Additionally, Range instances can be constructed by passing the bound types explicitly:

Range type	Method
Bounded on both ends	range(C, BoundType, C, BoundType)
Unbounded on top (`(a..+∞)` or `[a..+∞)`)	downTo(C, BoundType)
Unbounded on bottom (`(-∞..b)` or `(-∞..b]`)	upTo(C, BoundType)

Here, `BoundType` is an enum containing the values `CLOSED` and `OPEN`.

```
Range.downTo(4, boundType); // allows you to decide whether or not you want to
include 4
Range.range(1, CLOSED, 4, OPEN); // another way of writing Range.closedOpen(1, 4)
```

Operations

The fundamental operation of a `Range` is its `contains(C)` methods, which behaves exactly as you might expect. Additionally, a `Range` may be used as a `Predicate`, and used in functional idioms. Any `Range` also supports `containsAll(Iterable<? extends C>)`.

```
Range.closed(1, 3).contains(2); // returns true
Range.closed(1, 3).contains(4); // returns false
Range.lessThan(5).contains(5); // returns false
Range.closed(1, 4).containsAll(Ints.asList(1, 2, 3)); // returns true
```

Query Operations

To look at the endpoints of a range, `Range` exposes the following methods:

- `hasLowerBound()` and `hasUpperBound()` , which check if the range has the specified endpoints, or goes on "through infinity."
- `lowerBoundType()` and `upperBoundType()` return the `BoundType` for the corresponding endpoint, which can be either `CLOSED` or `OPEN` . If this range does not have the specified endpoint, the method throws an `IllegalStateException` .
- `lowerEndpoint()` and `upperEndpoint()` return the endpoints on the specified end, or throw an `IllegalStateException` if the range does not have the specified endpoint.
- `isEmpty()` tests if the range is empty, that is, it has the form `[a,a)` or `(a,a]` .

```
Range.closedOpen(4, 4).isEmpty(); // returns true
Range.openClosed(4, 4).isEmpty(); // returns true
Range.closed(4, 4).isEmpty(); // returns false
Range.open(4, 4).isEmpty(); // Range.open throws IllegalArgumentException

Range.closed(3, 10).lowerEndpoint(); // returns 3
Range.open(3, 10).lowerEndpoint(); // returns 3
Range.closed(3, 10).lowerBoundType(); // returns CLOSED
Range.open(3, 10).upperBoundType(); // returns OPEN
```

Interval Operations

`encloses`

The most basic relation on ranges is `encloses(Range)` , which is true if the bounds of the inner range do not extend outside the bounds of the outer range. This is solely dependent on comparisons between the endpoints!

- `[3..6]` encloses `[4..5]`
- `(3..6)` encloses `(3..6)`
- `[3..6]` encloses `[4..4)` (even though the latter is empty)
- `(3..6]` does not enclose `[3..6]`
- `[4..5]` does not enclose `(3..6)` **even though it contains every value contained by the latter range**, although use of discrete domains can address this (see below)
- `[3..6]` does not enclose `(1..1]` **even though it contains every value contained by the latter range**

`encloses` is a partial ordering.

Given this, `Range` provides the following operations:

`isConnected`

`Range.isConnected(Range)` , which tests if these ranges are *connected*. Specifically, `isConnected` tests if there is some range enclosed by both of these ranges, but this is equivalent to the mathematical definition that the union of the ranges must form a connected set (except in the special case of empty ranges).

`isConnected` is a reflexive, symmetric relation.

```
Range.closed(3, 5).isConnected(Range.open(5, 10)); // returns true
Range.closed(0, 9).isConnected(Range.closed(3, 4)); // returns true
Range.closed(0, 5).isConnected(Range.closed(3, 9)); // returns true
```

```
Range.open(3, 5).isConnected(Range.open(5, 10)); // returns false
Range.closed(1, 5).isConnected(Range.closed(6, 10)); // returns false
```

`intersection`

`Range.intersection(Range)` returns the maximal range enclosed by both this range and other (which exists iff these ranges are connected), or if no such range exists, throws an `IllegalArgumentException`.

`intersection` is a commutative, associative [operation][binary-operation].

```
Range.closed(3, 5).intersection(Range.open(5, 10)); // returns (5, 5]
Range.closed(0, 9).intersection(Range.closed(3, 4)); // returns [3, 4]
Range.closed(0, 5).intersection(Range.closed(3, 9)); // returns [3, 5]
Range.open(3, 5).intersection(Range.open(5, 10)); // throws IAE
Range.closed(1, 5).intersection(Range.closed(6, 10)); // throws IAE
```

`span`

`Range.span(Range)` returns the minimal range that encloses both this range and other. If the ranges are both connected, this is their union.

`span` is a commutative, associative and closed [operation][binary-operation].

```
Range.closed(3, 5).span(Range.open(5, 10)); // returns [3, 10)
Range.closed(0, 9).span(Range.closed(3, 4)); // returns [0, 9]
Range.closed(0, 5).span(Range.closed(3, 9)); // returns [0, 9]
Range.open(3, 5).span(Range.open(5, 10)); // returns (3, 10)
Range.closed(1, 5).span(Range.closed(6, 10)); // returns [1, 10]
```

Discrete Domains

Some types, but not all Comparable types, are *discrete*, meaning that ranges bounded on both sides can be enumerated.

In Guava, a `DiscreteDomain<C>` implements discrete operations for type `C`. A discrete domain always represents the entire set of values of its type; it cannot represent partial domains such as "prime integers", "strings of length 5," or "timestamps at midnight."

The `DiscreteDomain` class provides `DiscreteDomain` instances:

Type	DiscreteDomain
`Integer`	`integers()`
`Long`	`longs()`

Once you have a `DiscreteDomain`, you can use the following `Range` operations:

- `ContiguousSet.create(range, domain)`: view a `Range<C>` as an `ImmutableSortedSet<C>`, with a few extra operations thrown in. (Does not work for unbounded ranges, unless the type itself is bounded.)

- `canonical(domain)`: put ranges in a "canonical form." If `ContiguousSet.create(a, domain).equals(ContiguousSet.create(b, domain))` and `!a.isEmpty()`, then `a.canonical(domain).equals(b.canonical(domain))`. (This does *not*, however, imply `a.equals(b)`.)

```
ImmutableSortedSet<Integer> set = ContiguousSet.create(Range.open(1, 5),
DiscreteDomain.integers());
// set contains [2, 3, 4]

ContiguousSet.create(Range.greaterThan(0), DiscreteDomain.integers());
// set contains [1, 2, ..., Integer.MAX_VALUE]
```

Note that `ContiguousSet.create` does not *actually* construct the entire range, but instead returns a view of the range as a set.

Your Own DiscreteDomains

You can make your own `DiscreteDomain` objects, but there are several important aspects of the `DiscreteDomain` contract that you *must* remember.

- A discrete domain always represents the entire set of values of its type; it cannot represent partial domains such as "prime integers" or "strings of length 5." So you cannot, for example, construct a `DiscreteDomain` to view a set of days in a range, with a JODA `DateTime` that includes times up to the second: because this would not contain all elements of the type.
- A `DiscreteDomain` may be infinite -- a `BigInteger` `DiscreteDomain`, for example. In this case, you should use the default implementation of `minValue()` and `maxValue()`, which throw a `NoSuchElementException`. This forbids you from using the `ContiguousSet.create` method on an infinite range, however!

What if I need a `Comparator`?

We wanted to strike a very specific balance in `Range` between power and API complexity, and part of that involved not providing a `Comparator`-based interface: we don't need to worry about how ranges based on different comparators interact; the API signatures are all significantly simplified; things are just nicer.

On the other hand, if you think you want an arbitrary `Comparator`, you can do one of the following:

- Use a general `Predicate` and not `Range`. (Since `Range` implements the `Predicate` interface, you can use `Predicates.compose(range, function)` to get a `Predicate`.)
- Use a wrapper class around your objects that defines the desired ordering.