New Collection Types

Guava introduces a number of new collection types that are not in the JDK, but that we have found to be broadly useful. These are all designed to coexist happily with the JDK collections framework, without shoehorning things into the JDK collection abstractions.

As a general rule, the Guava collection implementations follow JDK interface contracts very precisely.

Multiset

The traditional Java idiom for e.g. counting how many times a word occurs in a document is something like:

```java
Map<String, Integer> counts = new HashMap<String, Integer>();
for (String word : words) {
  Integer count = counts.get(word);
  if (count == null) {
    counts.put(word, 1);
  } else {
    counts.put(word, count + 1);
  }
}
```

This is awkward, prone to mistakes, and doesn't support collecting a variety of useful statistics, like the total number of words. We can do better.

Guava provides a new collection type, `Multiset`, which supports adding multiples of elements. Wikipedia defines a multiset, in mathematics, as "a generalization of the notion of set in which members are allowed to appear more than once... In multisets, as in sets and in contrast to tuples, the order of elements is irrelevant: The multisets {a, a, b} and {a, b, a} are equal."

There are two main ways of looking at this:

- This is like an `ArrayList<E>` without an ordering constraint: ordering does not matter.
- This is like a `Map<E, Integer>`, with elements and counts.

Guava's `Multiset` API combines both ways of thinking about a `Multiset`, as follows:

- When treated as a normal `Collection`, `Multiset` behaves much like an unordered `ArrayList`:
  - Calling `add(E)` adds a single occurrence of the given element.
  - The `iterator()` of a Multiset iterates over every occurrence of every element.
  - The `size()` of a Multiset is the total number of all occurrences of all elements.

- The additional query operations, as well as the performance characteristics, are like what you'd expect from a Map<E, Integer>.
  - count(Object) returns the count associated with that element. For a HashMultiset, count is O(1), for a TreeMultiset, count is O(log n), etc.
  - entrySet() returns a Set<Multiset.Entry<E>> which works analogously to the entrySet of a Map.
  - elementSet() returns a Set<E> of the distinct elements of the multiset, like keySet() would for a Map.
  - The memory consumption of Multiset implementations is linear in the number of distinct elements.

Notably, Multiset is fully consistent with the contract of the Collection interface, save in rare cases with precedent in the JDK itself – specifically, TreeMultiset, like TreeSet, uses comparison for equality instead of Object.equals. In particular, Multiset.addAll(Collection) adds one occurrence of each element in the Collection for each time it appears, which is much more convenient than the for loop required by the Map approach above.

Method	Description
count(E)	Count the number of occurrences of an element that have been added to this multiset.
elementSet()	View the distinct elements of a Multiset<E> as a Set<E>.
entrySet()	Similar to Map.entrySet(), returns a Set<Multiset.Entry<E>>, containing entries supporting getElement() and getCount().
add(E, int)	Adds the specified number of occurrences of the specified element.
remove(E, int)	Removes the specified number of occurrences of the specified element.
setCount(E, int)	Sets the occurrence count of the specified element to the specified nonnegative value.

Method	Description
`size()`	Returns the total number of occurrences of all elements in the `Multiset`.

Multiset Is Not A Map

Note that `Multiset<E>` is *not* a `Map<E, Integer>`, though that might be part of a `Multiset` implementation. `Multiset` is a true `Collection` type, and satisfies all of the associated contractual obligations. Other notable differences include:

- A `Multiset<E>` has elements with positive counts only. No element can have negative counts, and values with count `0` are considered to not be in the multiset. They do not appear in the `elementSet()` or `entrySet()` view.
- `multiset.size()` returns the size of the collection, which is equal to the sum of the counts of all elements. For the number of distinct elements, use `elementSet().size()`. (So, for example, `add(E)` increases `multiset.size()` by one.)
- `multiset.iterator()` iterates over each occurrence of each element, so the length of the iteration is equal to `multiset.size()`.
- `Multiset<E>` supports adding elements, removing elements, or setting the count of elements directly. `setCount(elem, 0)` is equivalent to removing all occurrences of the element.
- `multiset.count(elem)` for an element not in the multiset always returns `0`.

Implementations

Guava provides many implementations of `Multiset`, which *roughly* correspond to JDK map implementations.

Map	Corresponding Multiset	Supports `null` elements
`HashMap`	`HashMultiset`	Yes
`TreeMap`	`TreeMultiset`	Yes
`LinkedHashMap`	`LinkedHashMultiset`	Yes
`ConcurrentHashMap`	`ConcurrentHashMultiset`	No
`ImmutableMap`	`ImmutableMultiset`	No

SortedMultiset

`SortedMultiset` is a variation on the `Multiset` interface that supports efficiently taking sub-multisets on specified ranges. For example, you could use `latencies.subMultiset(0, BoundType.CLOSED, 100,` `BoundType.OPEN).size()` to determine how many hits to your site had under 100ms latency, and then compare that to `latencies.size()` to determine the overall proportion.

`TreeMultiset` implements the `SortedMultiset` interface. At the time of writing, `ImmutableSortedMultiset` is still being tested for GWT compatibility.

Multimap

Every experienced Java programmer has, at one point or another, implemented a `Map<K, List<V>>` or `Map<K, Set<V>>`, and dealt with the awkwardness of that structure. For example, `Map<K, Set<V>>` is a typical way to represent an unlabeled directed graph. Guava's `Multimap` framework makes it easy to handle a mapping from keys to multiple values. A `Multimap` is a general way to associate keys with arbitrarily many values.

There are two ways to think of a Multimap conceptually: as a collection of mappings from single keys to single values:

```
a -> 1
a -> 2
a -> 4
b -> 3
c -> 5
```

or as a mapping from unique keys to collections of values:

```
a -> [1, 2, 4]
b -> [3]
c -> [5]
```

In general, the `Multimap` interface is best thought of in terms of the first view, but allows you to view it in either way with the `asMap()` view, which returns a `Map<K, Collection<V>>`. Most importantly, there is no such thing as a key which maps to an empty collection: a key either maps to at least one value, or it is simply not present in the `Multimap`.

You rarely use the `Multimap` interface directly, however; more often you'll use `ListMultimap` or `SetMultimap`, which map keys to a `List` or a `Set` respectively.

Construction

The most straight-forward way to create a `Multimap` is using `MultimapBuilder`, which allows you to configure how your keys and values should be represented. For example:

```
// creates a ListMultimap with tree keys and array list values
ListMultimap<String, Integer> treeListMultimap =
    MultimapBuilder.treeKeys().arrayListValues().build();
```

```java
// creates a SetMultimap with hash keys and enum set values
SetMultimap<Integer, MyEnum> hashEnumMultimap =
    MultimapBuilder.hashKeys().enumSetValues(MyEnum.class).build();
```

You may also choose to use the `create()` methods directly on the implementation classes, but that is lightly discouraged in favor of `MultimapBuilder`.

Modifying

`Multimap.get(key)` returns a *view* of the values associated with the specified key, even if there are none currently. For a `ListMultimap`, it returns a `List`, for a `SetMultimap`, it returns a `Set`.

Modifications write through to the underlying `Multimap`. For example,

```java
Set<Person> aliceChildren = childrenMultimap.get(alice);
aliceChildren.clear();
aliceChildren.add(bob);
aliceChildren.add(carol);
```

writes through to the underlying multimap.

Other ways of modifying the multimap (more directly) include:

Signature	Description	Equivalent
`put(K, V)`	Adds an association from the key to the value.	`multimap.get(key).add(value)`
`putAll(K, Iterable<V>)`	Adds associations from the key to each of the values in turn.	`Iterables.addAll(multimap.ge values)`
`remove(K, V)`	Removes one association from `key` to `value` and returns `true` if the multimap changed.	`multimap.get(key).remove(val`
`removeAll(K)`	Removes and returns all the values associated with the specified key. The returned collection may or may not be modifiable, but modifying it will not affect the multimap. (Returns the appropriate collection type.)	`multimap.get(key).clear()`
`replaceValues(K, Iterable<V>)`	Clears all the values associated with `key` and sets `key` to be associated with each of `values`. Returns the values that were previously associated with the key.	`multimap.get(key).clear(); Iterables.addAll(multimap.ge values)`

Views

`Multimap` also supports a number of powerful views.

- `asMap` views any `Multimap<K, V>` as a `Map<K, Collection<V>>`. The returned map supports `remove`, and changes to the returned collections write through, but the map does not support `put` or `putAll`. Critically, you can use `asMap().get(key)` when you want `null` on absent keys rather than a fresh, writable empty collection. (You can and should cast `asMap.get(key)` to the appropriate collection type – a `Set` for a `SetMultimap`, a `List` for a `ListMultimap` – but the type system does not allow `ListMultimap` to return `Map<K, List<V>>` here.)

- `entries` views the `Collection<Map.Entry<K, V>>` of all entries in the `Multimap`. (For a `SetMultimap`, this is a `Set`.)
- `keySet` views the distinct keys in the `Multimap` as a `Set`.
- `keys` views the keys of the `Multimap` as a `Multiset`, with multiplicity equal to the number of values associated to that key. Elements can be removed from the `Multiset`, but not added; changes will write through.
- `values()` views all the values in the `Multimap` as a "flattened" `Collection<V>`, all as one collection. This is similar to `Iterables.concat(multimap.asMap().values())` but returns a full `Collection` instead.

Multimap Is Not A Map

A `Multimap<K, V>` is *not* a `Map<K, Collection<V>>`, though such a map might be used in a `Multimap` implementation. Notable differences include:

- `Multimap.get(key)` always returns a non-null, possibly empty collection. This doesn't imply that the multimap spends any memory associated with the key, but instead, the returned collection is a view that allows you to add associations with the key if you like.
- If you prefer the more `Map`-like behavior of returning `null` for keys that aren't in the multimap, use the `asMap()` view to get a `Map<K, Collection<V>>`. (Or, to get a `Map<K,List<V>>` from a `ListMultimap`, use the static `Multimaps.asMap()` method. Similar methods exist for `SetMultimap` and `SortedSetMultimap`.)
- `Multimap.containsKey(key)` is true if and only if there are any elements associated with the specified key. In particular, if a key `k` was previously associated with one or more values which have since been removed from the multimap, `Multimap.containsKey(k)` will return false.
- `Multimap.entries()` returns all entries for all keys in the `Multimap`. If you want all key-collection entries, use `asMap().entrySet()`.
- `Multimap.size()` returns the number of entries in the entire multimap, not the number of distinct keys. Use `Multimap.keySet().size()` instead to get the number of distinct keys.

Implementations

`Multimap` provides a wide variety of implementations. Note it is generally prefered to create `Multimap` instances using `MultimapBuilder`.

Implementation	Keys behave like…	Values behave like..
`ArrayListMultimap`	`HashMap`	`ArrayList`
`HashMultimap`	`HashMap`	`HashSet`
`LinkedListMultimap *`	`LinkedHashMap``*`	`LinkedList``*`

Implementation	Keys behave like...	Values behave like..
`LinkedHashMultimap**`	`LinkedHashMap`	`LinkedHashSet`
`TreeMultimap`	`TreeMap`	`TreeSet`
`ImmutableListMultimap`	`ImmutableMap`	`ImmutableList`
`ImmutableSetMultimap`	`ImmutableMap`	`ImmutableSet`

Each of these implementations, except the immutable ones, support null keys and values.

`*` `LinkedListMultimap.entries()` preserves iteration order across non-distinct key values. See the link for details.

`**` `LinkedHashMultimap` preserves insertion order of entries, as well as the insertion order of keys, and the set of values associated with any one key.

Be aware that not all implementations are actually implemented as a `Map<K, Collection<V>>` with the listed implementations! (In particular, several `Multimap` implementations use custom hash tables to minimize overhead.)

If you need more customization, use `Multimaps.newMultimap(Map, Supplier<Collection>)` or the list and set versions to use a custom collection, list, or set implementation to back your multimap.

BiMap

The traditional way to map values back to keys is to maintain two separate maps and keep them both in sync, but this is bug-prone and can get extremely confusing when a value is already present in the map. For example:

```
Map<String, Integer> nameToId = Maps.newHashMap();
Map<Integer, String> idToName = Maps.newHashMap();

nameToId.put("Bob", 42);
idToName.put(42, "Bob");
// what happens if "Bob" or 42 are already present?
// weird bugs can arise if we forget to keep these in sync...
```

A `BiMap<K, V>` is a `Map<K, V>` that

- allows you to view the "inverse" `BiMap<V, K>` with `inverse()`
- ensures that values are unique, making `values()` a `Set`

`BiMap.put(key, value)` will throw an `IllegalArgumentException` if you attempt to map a key to an already-present value. If you wish to delete any preexisting entry with the specified value, use `BiMap.forcePut(key, value)` instead.

```
BiMap<String, Integer> userId = HashBiMap.create();
...
```

```java
String userForId = userId.inverse().get(id);
```

Implementations

Key-Value Map Impl	Value-Key Map Impl	Corresponding `BiMap`
`HashMap`	`HashMap`	`HashBiMap`
`ImmutableMap`	`ImmutableMap`	`ImmutableBiMap`
`EnumMap`	`EnumMap`	`EnumBiMap`
`EnumMap`	`HashMap`	`EnumHashBiMap`

Note: `BiMap` utilities like `synchronizedBiMap` live in `Maps`.

Table

```java
Table<Vertex, Vertex, Double> weightedGraph = HashBasedTable.create();
weightedGraph.put(v1, v2, 4);
weightedGraph.put(v1, v3, 20);
weightedGraph.put(v2, v3, 5);

weightedGraph.row(v1); // returns a Map mapping v2 to 4, v3 to 20
weightedGraph.column(v3); // returns a Map mapping v1 to 20, v2 to 5
```

Typically, when you are trying to index on more than one key at a time, you will wind up with something like `Map<FirstName, Map<LastName, Person>>`, which is ugly and awkward to use. Guava provides a new collection type, `Table`, which supports this use case for any "row" type and "column" type. `Table` supports a number of views to let you use the data from any angle, including

- `rowMap()`, which views a `Table<R, C, V>` as a `Map<R, Map<C, V>>`. Similarly, `rowKeySet()` returns a `Set<R>`.
- `row(r)` returns a non-null `Map<C, V>`. Writes to the `Map` will write through to the underlying `Table`.
- Analogous column methods are provided: `columnMap()`, `columnKeySet()`, and `column(c)`. (Column-based access is somewhat less efficient than row-based access.)
- `cellSet()` returns a view of the `Table` as a set of `Table.Cell<R, C, V>`. `Cell` is much like `Map.Entry`, but distinguishes the row and column keys.

Several `Table` implementations are provided, including:

- `HashBasedTable`, which is essentially backed by a `HashMap<R, HashMap<C, V>>`.
- `TreeBasedTable`, which is essentially backed by a `TreeMap<R, TreeMap<C, V>>`.

- `ImmutableTable`
- `ArrayTable`, which requires that the complete universe of rows and columns be specified at construction time, but is backed by a two-dimensional array to improve speed and memory efficiency when the table is dense. `ArrayTable` works somewhat differently from other implementations; consult the Javadoc for details.

ClassToInstanceMap

Sometimes, your map keys aren't all of the same type: they *are* types, and you want to map them to values of that type. Guava provides `ClassToInstanceMap` for this purpose.

In addition to extending the `Map` interface, `ClassToInstanceMap` provides the methods `T getInstance(Class<T>)` and `T putInstance(Class<T>, T)`, which eliminate the need for unpleasant casting while enforcing type safety.

`ClassToInstanceMap` has a single type parameter, typically named `B`, representing the upper bound on the types managed by the map. For example:

```
ClassToInstanceMap<Number> numberDefaults = MutableClassToInstanceMap.create();
numberDefaults.putInstance(Integer.class, Integer.valueOf(0));
```

Technically, `ClassToInstanceMap<B>` implements `Map<Class<? extends B>, B>` – or in other words, a map from subclasses of B to instances of B. This can make the generic types involved in `ClassToInstanceMap` mildly confusing, but just remember that `B` is always the upper bound on the types in the map – usually, `B` is just `Object`.

Guava provides implementations helpfully named `MutableClassToInstanceMap` and `ImmutableClassToInstanceMap`.

Important: Like any other `Map<Class, Object>`, a `ClassToInstanceMap` may contain entries for primitive types, and a primitive type and its corresponding wrapper type may map to different values.

RangeSet

A `RangeSet` describes a set of *disconnected, nonempty* ranges. When adding a range to a mutable `RangeSet`, any connected ranges are merged together, and empty ranges are ignored. For example:

```
RangeSet<Integer> rangeSet = TreeRangeSet.create();
rangeSet.add(Range.closed(1, 10)); // {[1, 10]}
rangeSet.add(Range.closedOpen(11, 15)); // disconnected range: {[1, 10], [11, 15)}
rangeSet.add(Range.closedOpen(15, 20)); // connected range; {[1, 10], [11, 20)}
rangeSet.add(Range.openClosed(0, 0)); // empty range; {[1, 10], [11, 20)}
rangeSet.remove(Range.open(5, 10)); // splits [1, 10]; {[1, 5], [10, 10], [11, 20)}
```

Note that to merge ranges like Range.closed(1, 10) and Range.closedOpen(11, 15), you must first preprocess ranges with Range.canonical(DiscreteDomain), e.g. with DiscreteDomain.integers().

NOTE: RangeSet is not supported under GWT, nor in the JDK 1.5 backport; RangeSet requires full use of the NavigableMap features in JDK 1.6.

Views

RangeSet implementations support an extremely wide range of views, including:

- complement(): views the complement of the RangeSet. complement is also a RangeSet, as it contains disconnected, nonempty ranges.
- subRangeSet(Range<C>): returns a view of the intersection of the RangeSet with the specified Range. This generalizes the headSet, subSet, and tailSet views of traditional sorted collections.
- asRanges(): views the RangeSet as a Set<Range<C>> which can be iterated over.
- asSet(DiscreteDomain<C>) (ImmutableRangeSet only): Views the RangeSet<C> as an ImmutableSortedSet<C>, viewing the elements in the ranges instead of the ranges themselves. (This operation is unsupported if the DiscreteDomain and the RangeSet are both unbounded above or both unbounded below.)

Queries

In addition to operations on its views, RangeSet supports several query operations directly, the most prominent of which are:

- contains(C): the most fundamental operation on a RangeSet, querying if any range in the RangeSet contains the specified element.
- rangeContaining(C): returns the Range which encloses the specified element, or null if there is none.
- encloses(Range<C>): straightforwardly enough, tests if any Range in the RangeSet encloses the specified range.
- span(): returns the minimal Range that encloses every range in this RangeSet.

RangeMap

RangeMap is a collection type describing a mapping from disjoint, nonempty ranges to values. Unlike RangeSet, RangeMap never "coalesces" adjacent mappings, even if adjacent ranges are mapped to the same values. For example:

```
RangeMap<Integer, String> rangeMap = TreeRangeMap.create();
rangeMap.put(Range.closed(1, 10), "foo"); // {[1, 10] => "foo"}
rangeMap.put(Range.open(3, 6), "bar"); // {[1, 3] => "foo", (3, 6) => "bar", [6, 10] => "fo
```

```java
rangeMap.put(Range.open(10, 20), "foo"); // {[1, 3] => "foo", (3, 6) => "bar", [6, 10] => "
rangeMap.remove(Range.closed(5, 11)); // {[1, 3] => "foo", (3, 5) => "bar", (11, 20) => "fo
```

Views

`RangeMap` provides two views:

- `asMapOfRanges()`: views the `RangeMap` as a `Map<Range<K>, V>`. This can be used, for example, to iterate over the `RangeMap`.
- `subRangeMap(Range<K>)` views the intersection of the `RangeMap` with the specified `Range` as a `RangeMap`. This generalizes the traditional `headMap`, `subMap`, and `tailMap` operations.