Collection Utilities

Any programmer with experience with the JDK Collections Framework knows and loves the utilities available in `java.util.Collections`. Guava provides many more utilities along these lines: static methods applicable to all collections. These are among the most popular and mature parts of Guava.

Methods corresponding to a particular interface are grouped in a relatively intuitive manner:

Interface	JDK or Guava?	Corresponding Guava utility class
`Collection`	JDK	`Collections2`
`List`	JDK	`Lists`
`Set`	JDK	`Sets`
`SortedSet`	JDK	`Sets`
`Map`	JDK	`Maps`
`SortedMap`	JDK	`Maps`
`Queue`	JDK	`Queues`
`Multiset`	Guava	`Multisets`
`Multimap`	Guava	`Multimaps`
`BiMap`	Guava	`Maps`
`Table`	Guava	`Tables`

Looking for transform, filter, and the like? That stuff is in our functional programming article, under functional idioms.

Static constructors

Before JDK 7, constructing new generic collections requires unpleasant code duplication:

```
List<TypeThatsTooLongForItsOwnGood> list = new ArrayList<TypeThatsTooLongForItsOwnGood>();
```

I think we can all agree that this is unpleasant. Guava provides static methods that use generics to infer the type on the right side:

```
List<TypeThatsTooLongForItsOwnGood> list = Lists.newArrayList();
Map<KeyType, LongishValueType> map = Maps.newLinkedHashMap();
```

To be sure, the diamond operator in JDK 7 makes this less of a hassle:

```
List<TypeThatsTooLongForItsOwnGood> list = new ArrayList<>();
```

But Guava goes further than this. With the factory method pattern, we can initialize collections with their starting elements very conveniently.

```
Set<Type> copySet = Sets.newHashSet(elements);
List<String> theseElements = Lists.newArrayList("alpha", "beta", "gamma");
```

Additionally, with the ability to name factory methods (Effective Java item 1), we can improve the readability of initializing collections to sizes:

```
List<Type> exactly100 = Lists.newArrayListWithCapacity(100);
List<Type> approx100 = Lists.newArrayListWithExpectedSize(100);
Set<Type> approx100Set = Sets.newHashSetWithExpectedSize(100);
```

The precise static factory methods provided are listed with their corresponding utility classes below.

Note: New collection types introduced by Guava don't expose raw constructors, or have initializers in the utility classes. Instead, they expose static factory methods directly, for example:

```
Multiset<String> multiset = HashMultiset.create();
```

Iterables

Whenever possible, Guava prefers to provide utilities accepting an `Iterable` rather than a `Collection`. Here at Google, it's not out of the ordinary to encounter a "collection" that isn't actually stored in main memory, but is being gathered from a database, or from another data center, and can't support operations like `size()` without actually grabbing all of the elements.

As a result, many of the operations you might expect to see supported for all collections can be found in `Iterables`. Additionally, most `Iterables` methods have a corresponding version in `Iterators` that accepts the raw iterator.

The overwhelming majority of operations in the `Iterables` class are *lazy*: they only advance the backing iteration when absolutely necessary. Methods that themselves return `Iterables` return lazily computed views, rather than explicitly constructing a collection in memory.

As of Guava 12, `Iterables` is supplemented by the `FluentIterable` class, which wraps an `Iterable` and provides a "fluent" syntax for many of these operations.

The following is a selection of the most commonly used utilities, although many of the more "functional" methods in `Iterables` are discussed in Guava functional idioms.

General

Method	Description	See Also
`concat(Iterable<Iterable>)`	Returns a lazy view of the concatenation of several iterables.	`concat(Iterable...)`

Method	Description	See Also
`frequency(Iterable, Object)`	Returns the number of occurrences of the object.	Compare `Collections.frequency(Collect Object)`; see `Multiset`
`partition(Iterable, int)`	Returns an unmodifiable view of the iterable partitioned into chunks of the specified size.	`Lists.partition(List, int)`, `paddedPartition(Iterable, int)`
`getFirst(Iterable, T default)`	Returns the first element of the iterable, or the default value if empty.	Compare `Iterable.iterator().next()`, `FluentIterable.first()`
`getLast(Iterable)`	Returns the last element of the iterable, or fails fast with a `NoSuchElementException` if it's empty.	`getLast(Iterable, T default)`, `FluentIterable.last()`
`elementsEqual(Iterable, Iterable)`	Returns true if the iterables have the same elements in the same order.	Compare `List.equals(Object)`
`unmodifiableIterable(Iterable)`	Returns an unmodifiable view of the iterable.	Compare `Collections.unmodifiableColle`
`limit(Iterable, int)`	Returns an `Iterable` returning at most the specified number of elements.	`FluentIterable.limit(int)`
`getOnlyElement(Iterable)`	Returns the only element in `Iterable`. Fails fast if the iterable is empty or has multiple elements.	`getOnlyElement(Iterable, T default)`

```java
Iterable<Integer> concatenated = Iterables.concat(
  Ints.asList(1, 2, 3),
  Ints.asList(4, 5, 6));
// concatenated has elements 1, 2, 3, 4, 5, 6

String lastAdded = Iterables.getLast(myLinkedHashSet);

String theElement = Iterables.getOnlyElement(thisSetIsDefinitelyASingleton);
  // if this set isn't a singleton, something is wrong!
```

Collection-Like

Typically, collections support these operations naturally on other collections, but not on iterables.

Each of these operations delegates to the corresponding `Collection` interface method when the input is actually a `Collection`. For example, if

`Iterables.size` is passed a `Collection`, it will call the `Collection.size` method instead of walking through the iterator.

Method	Analogous `Collection` method	`FluentIterable` equivalent
`addAll(Collection addTo, Iterable toAdd)`	`Collection.addAll(Collection)`	
`contains(Iterable, Object)`	`Collection.contains(Object)`	`FluentIterable.contains(Object)`
`removeAll(Iterable removeFrom, Collection toRemove)`	`Collection.removeAll(Collection)`	
`retainAll(Iterable removeFrom, Collection toRetain)`	`Collection.retainAll(Collection)`	
`size(Iterable)`	`Collection.size()`	`FluentIterable.size()`
`toArray(Iterable, Class)`	`Collection.toArray(T[])`	`FluentIterable.toArray(Class)`
`isEmpty(Iterable)`	`Collection.isEmpty()`	`FluentIterable.isEmpty()`
`get(Iterable, int)`	`List.get(int)`	`FluentIterable.get(int)`
`toString(Iterable)`	`Collection.toString()`	`FluentIterable.toString()`

FluentIterable

Besides the methods covered above and in the functional idioms [article] functional, `FluentIterable` has a few convenient methods for copying into an immutable collection:

Result Type	Method
`ImmutableList`	`toImmutableList()`
`ImmutableSet`	`toImmutableSet()`
`ImmutableSortedSet`	`toImmutableSortedSet(Comparator)`

Lists

In addition to static constructor methods and functional programming methods, `Lists` provides a number of valuable utility methods on `List` objects.

Method	Description
`partition(List, int)`	Returns a view of the underlying list, partitioned into chunks of the specified size.

Method	Description
`reverse(List)`	Returns a reversed view of the specified list. *Note*: if the list is immutable, consider `ImmutableList.reverse()` instead.

```
List<Integer> countUp = Ints.asList(1, 2, 3, 4, 5);
List<Integer> countDown = Lists.reverse(theList); // {5, 4, 3, 2, 1}

List<List<Integer>> parts = Lists.partition(countUp, 2); // {{1, 2}, {3, 4}, {5}}
```

Static Factories

`Lists` provides the following static factory methods:

Implementation	Factories
`ArrayList`	basic, with elements, from `Iterable`, with exact capacity, with expected size, from `Iterator`
`LinkedList`	basic, from `Iterable`

Comparators

Finding the minimum or maximum of some elements

A seemingly simple task (finding the min or max of some elements) is complicated by the desire to minimize allocations, boxing, and APIs living in a variety of locations. The table below summarizes the best practices for this task.

Only the `max()` solution is shown in the table below, but the same advice applies for finding a `min()`.

What you're comparing	Exactly 2 instances	More than 2 instances
unboxed numeric primitives(e.g., `long`, `int`, `double`, or `float`)	`Math.max(a, b)`	`Longs.max(a, b, c)`,`Ints.max(a, b, c)`,etc.
`Comparable` instances(e.g., `Duration`, `String`, `Long`, etc.)	`Comparators.max(a, b)`	`Collections.max(asList(a, b, c))`

What you're comparing	Exactly 2 instances	More than 2 instances
using a custom Comparator(e.g., MyType with myComparator)	Comparators.max(a, b, cmp)	Collections.max(asList(a, b, c), cmp)

Note: We recommend static importing all of the methods involved in these solutions to simplify your code (e.g., prefer max(asList(a, b, c)) over Collections.max(Arrays.asList(a, b, c))).

Sets

The Sets utility class includes a number of spicy methods.

Set-Theoretic Operations

We provide a number of standard set-theoretic operations, implemented as views over the argument sets. These return a SetView, which can be used:

- as a Set directly, since it implements the Set interface
- by copying it into another mutable collection with copyInto(Set)
- by making an immutable copy with immutableCopy()

Method | :——————————— | union(Set, Set) | intersection(Set, Set) | difference(Set, Set) | symmetricDifference(Set, Set) |

For example:

```
Set<String> wordsWithPrimeLength = ImmutableSet.of("one", "two", "three", "six", "seven", "e
Set<String> primes = ImmutableSet.of("two", "three", "five", "seven");

SetView<String> intersection = Sets.intersection(primes, wordsWithPrimeLength); // contains
// I can use intersection as a Set directly, but copying it can be more efficient if I use 
return intersection.immutableCopy();
```

Other Set Utilities

Method	Description	See Also
cartesianProduct(List<Set>)	Returns every possible list that can be obtained by choosing one element from each set.	cartesianProduct(Set...)
powerSet(Set)	Returns the set of subsets of the specified set.	

```
Set<String> animals = ImmutableSet.of("gerbil", "hamster");
Set<String> fruits = ImmutableSet.of("apple", "orange", "banana");

Set<List<String>> product = Sets.cartesianProduct(animals, fruits);
```

```
// {{"gerbil", "apple"}, {"gerbil", "orange"}, {"gerbil", "banana"},
//  {"hamster", "apple"}, {"hamster", "orange"}, {"hamster", "banana"}}

Set<Set<String>> animalSets = Sets.powerSet(animals);
// {{}, {"gerbil"}, {"hamster"}, {"gerbil", "hamster"}}
```

Static Factories

`Sets` provides the following static factory methods:

Implementation	Factories
`HashSet`	basic, with elements, from `Iterable`, with expected size, from `Iterator`
`LinkedHashSet`	basic, from `Iterable`, with expected size
`TreeSet`	basic, with `Comparator`, from `Iterable`

Maps

`Maps` has a number of cool utilities that deserve individual explanation.

`uniqueIndex`

`Maps.uniqueIndex(Iterable, Function)` addresses the common case of having a bunch of objects that each have some unique attribute, and wanting to be able to look up those objects based on that attribute.

Let's say we have a bunch of strings that we know have unique lengths, and we want to be able to look up the string with some particular length.

```
ImmutableMap<Integer, String> stringsByIndex = Maps.uniqueIndex(strings, new Function<String
    public Integer apply(String string) {
      return string.length();
    }
  });
```

If indices are *not* unique, see `Multimaps.index` below.

`difference`

`Maps.difference(Map, Map)` allows you to compare all the differences between two maps. It returns a `MapDifference` object, which breaks down the Venn diagram into:

Method	Description
`entriesInCommon()`	The entries which are in both maps, with both matching keys and values.
`entriesDiffering()`	The entries with the same keys, but differing values. The values in this map are of type `MapDifference.ValueDifference`, which lets you look at the left and right values.
`entriesOnlyOnLeft()`	Returns the entries whose keys are in the left but not in the right map.
`entriesOnlyOnRight()`	Returns the entries whose keys are in the right but not in the left map.

```
Map<String, Integer> left = ImmutableMap.of("a", 1, "b", 2, "c", 3);
Map<String, Integer> right = ImmutableMap.of("b", 2, "c", 4, "d", 5);
MapDifference<String, Integer> diff = Maps.difference(left, right);

diff.entriesInCommon(); // {"b" => 2}
diff.entriesDiffering(); // {"c" => (3, 4)}
diff.entriesOnlyOnLeft(); // {"a" => 1}
diff.entriesOnlyOnRight(); // {"d" => 5}
```

`BiMap` utilities

The Guava utilities on `BiMap` live in the `Maps` class, since a `BiMap` is also a `Map`.

`BiMap` utility	Corresponding `Map` utility
`synchronizedBiMap(BiMap)`	`Collections.synchronizedMap(Map)`
`unmodifiableBiMap(BiMap)`	`Collections.unmodifiableMap(Map)`

Static Factories `Maps` provides the following static factory methods.

Implementation	Factories
`HashMap`	basic, from `Map`, with expected size
`LinkedHashMap`	basic, from `Map`
`TreeMap`	basic, from `Comparator`, from `SortedMap`
`EnumMap`	from `Class`, from `Map`
`ConcurrentMap`	basic
`IdentityHashMap`	basic

Multisets

Standard `Collection` operations, such as `containsAll`, ignore the count of elements in the multiset, and only care about whether elements are in the multiset at all, or not. `Multisets` provides a number of operations that take into account element multiplicities in multisets.

Method	Explanation	Difference from `Collection` method
`containsOccurrences(Multiset sup, Multiset sub)`	Returns `true` if `sub.count(o) <= super.count(o)` for all `o`.	`Collection.containsAll` ignores counts, and only tests whether elements are contained at all.
`removeOccurrences(Multiset removeFrom, Multiset toRemove)`	Removes one occurrence in `removeFrom` for each occurrence of an element in `toRemove`.	`Collection.removeAll` removes all occurrences of any element that occurs even once in `toRemove`.
`retainOccurrences(Multiset removeFrom, Multiset toRetain)`	Guarantees that `removeFrom.count(o) <= toRetain.count(o)` for all `o`.	`Collection.retainAll` keeps all occurrences of elements that occur even once in `toRetain`.

Method	Explanation	Difference from `Collection` method
`intersection(Multiset, Multiset)`	Returns a view of the intersection of two multisets; a nondestructive alternative to `retainOccurrences`.	Has no analogue.

```
Multiset<String> multiset1 = HashMultiset.create();
multiset1.add("a", 2);

Multiset<String> multiset2 = HashMultiset.create();
multiset2.add("a", 5);

multiset1.containsAll(multiset2); // returns true: all unique elements are contained,
  // even though multiset1.count("a") == 2 < multiset2.count("a") == 5
Multisets.containsOccurrences(multiset1, multiset2); // returns false

Multisets.removeOccurrences(multiset2, multiset1); // multiset2 now contains 3 occurrences o

multiset2.removeAll(multiset1); // removes all occurrences of "a" from multiset2, even thoug
multiset2.isEmpty(); // returns true
```

Other utilities in `Multisets` include:

Method	Description
`copyHighestCountFirst(Multiset)`	Returns an immutable copy of the multiset that iterates over elements in descending frequency order.
`unmodifiableMultiset(Multiset)`	Returns an unmodifiable view of the multiset.
`unmodifiableSortedMultiset(SortedMultiset)`	Returns an unmodifiable view of the sorted multiset.

```java
Multiset<String> multiset = HashMultiset.create();
multiset.add("a", 3);
multiset.add("b", 5);
multiset.add("c", 1);

ImmutableMultiset<String> highestCountFirst = Multisets.copyHighestCountFirst(multiset);

// highestCountFirst, like its entrySet and elementSet, iterates over the elements in order
```

Multimaps

`Multimaps` provides a number of general utility operations that deserve individual explanation.

`index`

The cousin to `Maps.uniqueIndex`, `Multimaps.index(Iterable, Function)` answers the case when you want to be able to look up all objects with some particular attribute in common, which is not necessarily unique.

Let's say we want to group strings based on their length.

```java
ImmutableSet<String> digits = ImmutableSet.of(
    "zero", "one", "two", "three", "four",
    "five", "six", "seven", "eight", "nine");
Function<String, Integer> lengthFunction = new Function<String, Integer>() {
  public Integer apply(String string) {
    return string.length();
  }
};
ImmutableListMultimap<Integer, String> digitsByLength = Multimaps.index(digits, lengthFuncti
/*
 * digitsByLength maps:
 *  3 => {"one", "two", "six"}
 *  4 => {"zero", "four", "five", "nine"}
 *  5 => {"three", "seven", "eight"}
 */
```

`invertFrom`

Since `Multimap` can map many keys to one value, and one key to many values, it can be useful to invert a `Multimap`. Guava provides `invertFrom(Multimap toInvert, Multimap dest)` to let you do this, without choosing an implementation for you.

NOTE: If you are using an `ImmutableMultimap`, consider `ImmutableMultimap.inverse()` instead.

```java
ArrayListMultimap<String, Integer> multimap = ArrayListMultimap.create();
multimap.putAll("b", Ints.asList(2, 4, 6));
multimap.putAll("a", Ints.asList(4, 2, 1));
multimap.putAll("c", Ints.asList(2, 5, 3));

TreeMultimap<Integer, String> inverse = Multimaps.invertFrom(multimap, TreeMultimap.<Integer
// note that we choose the implementation, so if we use a TreeMultimap, we get results in o
/*
 * inverse maps:
 *  1 => {"a"}
 *  2 => {"a", "b", "c"}
 *  3 => {"c"}
 *  4 => {"a", "b"}
 *  5 => {"c"}
 *  6 => {"b"}
 */
```

`forMap`

Need to use a `Multimap` method on a `Map`? `forMap(Map)` views a `Map` as a `SetMultimap`. This is particularly useful, for example, in combination with `Multimaps.invertFrom`.

```java
Map<String, Integer> map = ImmutableMap.of("a", 1, "b", 1, "c", 2);
SetMultimap<String, Integer> multimap = Multimaps.forMap(map);
// multimap maps ["a" => {1}, "b" => {1}, "c" => {2}]
Multimap<Integer, String> inverse = Multimaps.invertFrom(multimap, HashMultimap.<Integer, St
// inverse maps [1 => {"a", "b"}, 2 => {"c"}]
```

Wrappers

`Multimaps` provides the traditional wrapper methods, as well as tools to get custom `Multimap` implementations based on `Map` and `Collection` implementations of your choice.

Multimap type	Unmodifiable	Synchronized	Custom
`Multimap`	`unmodifiableMultimap`	`synchronizedMultimap`	`newMultimap`
`ListMultimap`	`unmodifiableListMultimap`	`synchronizedListMultimap`	`newListMultimap`
`SetMultimap`	`unmodifiableSetMultimap`	`synchronizedSetMultimap`	`newSetMultimap`
`SortedSetMultimap`	`unmodifiableSortedSetMultimap`	`synchronizedSortedSetMultimap`	`newSortedSetMultimap`

The custom `Multimap` implementations let you specify a particular implementation that should be used in the returned `Multimap`. Caveats include:

- The multimap assumes complete ownership over of map and the lists returned by factory. Those objects should not be manually updated, they

should be empty when provided, and they should not use soft, weak, or phantom references.
- **No guarantees are made** on what the contents of the `Map` will look like after you modify the `Multimap`.
- The multimap is not threadsafe when any concurrent operations update the multimap, even if map and the instances generated by factory are. Concurrent read operations will work correctly, though. Work around this with the `synchronized` wrappers if necessary.
- The multimap is serializable if map, factory, the lists generated by factory, and the multimap contents are all serializable.
- The collections returned by `Multimap.get(key)` are *not* of the same type as the collections returned by your `Supplier`, though if you supplier returns `RandomAccess` lists, the lists returned by `Multimap.get(key)` will also be random access.

Note that the custom `Multimap` methods expect a `Supplier` argument to generate fresh new collections. Here is an example of writing a `ListMultimap` backed by a `TreeMap` mapping to `LinkedList`.

```
ListMultimap<String, Integer> myMultimap = Multimaps.newListMultimap(
  Maps.<String, Collection<Integer>>newTreeMap(),
  new Supplier<LinkedList<Integer>>() {
    public LinkedList<Integer> get() {
      return Lists.newLinkedList();
    }
  });
```

Tables

The `Tables` class provides a few handy utilities.

`customTable`

Comparable to the `Multimaps.newXXXMultimap(Map, Supplier)` utilities, `Tables.newCustomTable(Map, Supplier<Map>)` allows you to specify a `Table` implementation using whatever row or column map you like.

```
// use LinkedHashMaps instead of HashMaps
Table<String, Character, Integer> table = Tables.newCustomTable(
  Maps.<String, Map<Character, Integer>>newLinkedHashMap(),
  new Supplier<Map<Character, Integer>> () {
    public Map<Character, Integer> get() {
      return Maps.newLinkedHashMap();
    }
  });
```

`transpose`

The `transpose(Table<R, C, V>)` method allows you to view a `Table<R, C, V>` as a `Table<C, R, V>`.

Wrappers

These are the familiar unmodifiability wrappers you know and love. Consider, however, using `ImmutableTable` instead in most cases.

- `unmodifiableTable`
- `unmodifiableRowSortedTable`