Immutable Collections

Example

```
public static final ImmutableSet<String> COLOR_NAMES = ImmutableSet.of(
  "red",
  "orange",
  "yellow",
  "green",
  "blue",
  "purple");

class Foo {
  final ImmutableSet<Bar> bars;
  Foo(Set<Bar> bars) {
    this.bars = ImmutableSet.copyOf(bars); // defensive copy!
  }
}
```

Why?

Immutable objects have many advantages, including:

- Safe for use by untrusted libraries.
- Thread-safe: can be used by many threads with no risk of race conditions.
- Doesn't need to support mutation, and can make time and space savings with that assumption. All immutable collection implementations are more memory-efficient than their mutable siblings. (analysis)
- Can be used as a constant, with the expectation that it will remain fixed.

Making immutable copies of objects is a good defensive programming technique. Guava provides simple, easy-to-use immutable versions of each standard `Collection` type, including Guava's own `Collection` variations.

The JDK provides `Collections.unmodifiableXXX` methods, but in our opinion, these can be

- unwieldy and verbose; unpleasant to use everywhere you want to make defensive copies
- unsafe: the returned collections are only truly immutable if nobody holds a reference to the original collection
- inefficient: the data structures still have all the overhead of mutable collections, including concurrent modification checks, extra space in hash tables, etc.

When you don't expect to modify a collection, or expect a collection to remain constant, it's a good practice to defensively copy it into an immutable collection.

Important: Each of the Guava immutable collection implementations *rejects null values.* We did an exhaustive study on Google's internal code base that indicated that `null` elements were allowed in collections about 5% of the time, and the other 95% of cases were best served by failing fast on nulls. If you need to use null values, consider using `Collections.unmodifiableList` and its friends on a collection implementation that permits null. More detailed suggestions can be found here.

How?

An `ImmutableXXX` collection can be created in several ways:

- using the `copyOf` method, for example, `ImmutableSet.copyOf(set)`
- using the `of` method, for example, `ImmutableSet.of("a", "b", "c")` or `ImmutableMap.of("a", 1, "b", 2)`
- using a `Builder`, for example,

```
public static final ImmutableSet<Color> GOOGLE_COLORS =
   ImmutableSet.<Color>builder()
       .addAll(WEBSAFE_COLORS)
       .add(new Color(0, 191, 255))
       .build();
```

Except for sorted collections, **order is preserved from construction time**. For example,

```
ImmutableSet.of("a", "b", "c", "a", "d", "b")
```

will iterate over its elements in the order "a", "b", "c", "d".

`copyOf` is smarter than you think

It is useful to remember that `ImmutableXXX.copyOf` attempts to avoid copying the data when it is safe to do so -- the exact details are unspecified, but the implementation is typically "smart". For example,

```
ImmutableSet<String> foobar = ImmutableSet.of("foo", "bar", "baz");
thingamajig(foobar);

void thingamajig(Collection<String> collection) {
   ImmutableList<String> defensiveCopy = ImmutableList.copyOf(collection);
   ...
}
```

In this code, `ImmutableList.copyOf(foobar)` will be smart enough to just return `foobar.asList()`, which is a constant-time view of the `ImmutableSet`.

As a general heuristic, `ImmutableXXX.copyOf(ImmutableCollection)` tries to avoid a linear-time copy if

- it's possible using the underlying data structures in constant time. For example, `ImmutableSet.copyOf(ImmutableList)` can't be done in constant time.
- it wouldn't cause memory leaks -- for example, if you have `ImmutableList<String> hugeList`, and you do `ImmutableList.copyOf(hugeList.subList(0, 10))`, an explicit copy is performed, so as to avoid accidentally holding on to references in `hugeList` that aren't needed.
- it won't change semantics -- so `ImmutableSet.copyOf(myImmutableSortedSet)` will perform an explicit copy, because the `hashCode()` and `equals` used by `ImmutableSet` have different semantics from the comparator-based behavior of `ImmutableSortedSet`.

This helps minimize the performance overhead of good defensive programming style.

`asList`

All immutable collections provide an `ImmutableList` view via `asList()`, so -- for example -- even if you have data stored as an `ImmutableSortedSet`, you can get the `k` th smallest element with `sortedSet.asList().get(k)`.

The returned `ImmutableList` is frequently -- not always, but frequently -- a constant-overhead view, rather than an explicit copy. That said, it's often smarter than your average `List` -- for example, it'll use the efficient `contains` methods of the backing collection.

Details

Where?

Interface	JDK or Guava?	Immutable Version
`Collection`	JDK	`ImmutableCollection`
`List`	JDK	`ImmutableList`
`Set`	JDK	`ImmutableSet`
`SortedSet`/`NavigableSet`	JDK	`ImmutableSortedSet`
`Map`	JDK	`ImmutableMap`
`SortedMap`	JDK	`ImmutableSortedMap`
`Multiset`	Guava	`ImmutableMultiset`
`SortedMultiset`	Guava	`ImmutableSortedMultiset`
`Multimap`	Guava	`ImmutableMultimap`
`ListMultimap`	Guava	`ImmutableListMultimap`
`SetMultimap`	Guava	`ImmutableSetMultimap`
`BiMap`	Guava	`ImmutableBiMap`
`ClassToInstanceMap`	Guava	`ImmutableClassToInstanceMap`
`Table`	Guava	`ImmutableTable`