

A decorative header at the top of the slide features four overlapping spheres: a green one on the left, and three others (blue, red, and yellow) on the right. A thin black horizontal line is positioned below the spheres.

# The Google Collections Library (for Java)

Kevin Bourrillion, Google, Inc.  
SV-GTUG 2008-08-06



# In a nutshell

- An open-source (Apache 2) library
- <http://google-collections.googlecode.com>
- Requires JDK 1.5
- Pre-release snapshot available, 0.9 release coming
- Not API-frozen until release 1.0 (no timetable)
- But is *widely* used in production at Google
- Developers: Jared Levy and myself, with a *lot* of help from our friends



# Overview

---

1. In a nutshell
- 2. Immutable Collections**
3. Multisets
4. Multimaps
5. Other new types/impls
6. Static utilities
7. Other stuff
8. Q & A

Questions also welcome throughout!



# Immutable Collections

- JDK has `Collections.unmodifiableFoo()` wrappers
- Unmodifiable = *you* can't change it
- Immutable = it can never change, no matter what
  - (externally-visible state, that is)
- Immutability is tasty!
  - See Effective Java for some of the many reasons



# Immutable Collections (2)

---

We provide:

- ImmutableList
- ImmutableSet
- ImmutableSortedSet
- ImmutableMap
- ImmutableSortedMap (one day)

Brand-new, standalone implementations



# Immutable vs. unmodifiable

The JDK wrappers are still useful for unmodifiable *views* of changing data. But for most purposes, use ours:

- Immutability guarantee!
- Very easy to use
  - (we'll show you on the following slides)
- Slightly faster
- Use less memory
  - Sometimes *far* less (ImmutableSet, factor of 2-3x)



# Constant sets: Before, v1

```
public static final Set<Integer> LUCKY_NUMBERS;  
static {  
    Set<Integer> set = new LinkedHashSet<Integer>();  
    set.add(4);  
    set.add(8);  
    set.add(15);  
    set.add(16);  
    set.add(23);  
    set.add(42);  
    LUCKY_NUMBERS = Collections.unmodifiableSet(set);  
}
```



# Constant sets: Before, v2

```
public static final Set<Integer> LUCKY_NUMBERS
    = Collections.unmodifiableSet(
        new LinkedHashSet<Integer>(
            Arrays.asList(4, 8, 15, 16, 23, 42)));
```

- A little nicer.
- But uses four different classes! Something's weird.



# Constant sets: After

```
public static final ImmutableSet<Integer> LUCKY_NUMBERS  
    = ImmutableSet.of(4, 8, 15, 16, 23, 42);
```

- Now we just say exactly what we mean.
- And get performance benefits as well!
- We're using just one class (it implements Set)
- of() method name inspired by java.util.EnumSet



# Constant maps: Before

```
public static final Map<String, Integer> ENGLISH_TO_INT;  
static {  
    Map<String, Integer> map  
        = new LinkedHashMap<String, Integer>();  
    map.put("four", 4);  
    map.put("eight", 8);  
    map.put("fifteen", 15);  
    map.put("sixteen", 16);  
    map.put("twenty-three", 23);  
    map.put("forty-two", 42);  
    ENGLISH_TO_INT = Collections.unmodifiableMap(map);  
}
```



# Constant maps: After

```
public static final ImmutableMap<String, Integer>
    ENGLISH_TO_INT = ImmutableMap
        .with("four", 4)
        .with("eight", 8)
        .with("fifteen", 15)
        .with("sixteen", 16)
        .with("twenty-three", 23)
        .with("forty-two", 42)
        .build();
```



# Defensive copies: Before

```
private final Set<Integer> luckyNumbers;  
public Dharma(Set<Integer> numbers) {  
    luckyNumbers = Collections.unmodifiableSet(  
        new LinkedHashSet<Integer>(numbers));  
}  
public Set<Integer> getLuckyNumbers() {  
    return luckyNumbers;  
}
```

- Copy on the way in
- Wrap in unmodifiable on the way in or the way out

# Defensive copies: After

```
private final ImmutableSet<Integer> luckyNumbers;  
public Dharma(Set<Integer> numbers) {  
    luckyNumbers = ImmutableSet.copyOf(numbers);  
}  
public ImmutableSet<Integer> getLuckyNumbers() {  
    return luckyNumbers;  
}
```

- Type, not just implementation
- What if you forget?
- Note: `copyOf()` cheats!

# Immutable Collections: more examples

## Sets:

```
static final ImmutableSet<Country> BETA_COUNTRIES = ...  
  
    ImmutableSet.of();  
    ImmutableSet.of(a);  
    ImmutableSet.of(a, b, c);  
    ImmutableSet.copyOf(someIterator);  
    ImmutableSet.copyOf(someIterable);
```

## Small maps:

```
static final ImmutableMap<Integer, String> MAP  
    = ImmutableMap.of(1, "one", 2, "two");
```



# Immutable Collections: caveats

---

- These collections are null-hostile
  - In 95%+ of cases, this is what you want
  - In other cases, fine workarounds exist
  - This aligns with recent work on JDK collections
    - (and it's a little faster this way)
    - (and keeps the implementation simpler)
- Mutable elements can sometimes lead to confusion
  - The resulting object won't be "deeply immutable"

# Immutable Collections: summary

---

- In the past, we'd ask, "does this *need* to be immutable?"
- Now we ask, "does it *need* to be mutable?"

# Overview

---

1. In a nutshell
2. Immutable Collections
- 3. Multisets**
4. Multimaps
5. Other new types/impls
6. Static utilities
7. Other stuff
8. Q & A

Questions also welcome throughout!



# Collection behavior

When you have "a bunch of foos", use a Collection -- but what kind?

- Can it have duplicates?
- Is ordering significant? (for equals())
- Iteration order
  - insertion-ordered? comparator-ordered? user-ordered?
  - something else well-defined?
  - or it just doesn't matter?

In general, the first two determine the interface type, and the third tends to influence your choice of implementation.



# List vs. Set

Set: unordered equality, no duplicates.

List: ordered equality, can have duplicates.

	Ordered?	
	Y	N
Dups?	+-----+	+-----+
Y	List	?
	+-----+	+-----+
N	?	Set
	+-----+	+-----+

# List vs. Set... and then some

Multiset: unordered equality, can have duplicates.

	Ordered?	
	Y	N
Dups?	+-----+	+-----+
Y	List	<b>Multiset!</b>
N	(UniqueList)	Set

(UniqueList *might* appear in our library one day.)



# When to use a Multiset?

- "I kinda want a Set except I can have duplicates"
  - card game example
  - changing to List sacrifices contains() performance
- "Are these Lists equal, ignoring order?"
  - write a utility method for this?
- Histograms
  - "What distinct tags am I using on my blog, and how many times do I use each one?"

Multiset performance varies by the number of distinct elements, not total size.



# Multiset: tags example, before

```
Map<String, Integer> tags
    = new HashMap<String, Integer>();
for (BlogPost post : getAllBlogPosts()) {
    for (String tag : post.getTags()) {
        int value = tags.containsKey(tag) ? tags.get(tag) : 0;
        tags.put(tag, value + 1);
    }
}
```

- **distinct tags:** `tags.keySet()`
- **count for "java" tag:** `tags.containsKey("java") ? tags.get("java") : 0;`
- **total count:** `// oh crap...`



# Multiset: tags example, after

```
Multiset<String> tags = HashMultiset.create();  
for (BlogPost post : getAllBlogPosts()) {  
    tags.addAll(post.getTags());  
}
```

(... this space intentionally left blank ...)

- **distinct tags:** `tags.elementSet();`
- **count for "java" tag:** `tags.count("java");`
- **total count:** `tags.size();`



# Multiset: tags example, after after

- What if you need to remove/decrement?
  - Don't accidentally go negative
  - Don't forget to prune!
  - (Or just use a Multiset.)
- What about concurrency?
  - Lock the entire map just to add one tag?
  - (Or just use our ConcurrentMultiset.)
- When you use a powerful library, your code can easily evolve.

# Multiset API

Everything from Collection, plus:

```
int count(Object element);
```

```
int add(E element, int occurrences);
```

```
boolean remove(Object element, int occurrences);
```

```
int setCount(E element, int newCount);
```

```
boolean setCount(E e, int oldCount, int newCount);
```



# Multiset implementations

---

- ImmutableMultiset
- HashMultiset
- LinkedHashMultiset
- TreeMultiset
- EnumMultiset
- ConcurrentMultiset

# Overview

---

1. In a nutshell
2. Immutable Collections
3. Multisets
- 4. Multimaps**
5. Other new types/impls
6. Static utilities
7. Other stuff
8. Q & A

Questions also welcome throughout!



# Multimaps, before

Ever done this?

```
Map<Salesperson, List<Sale>> map
    = new HashMap<Salesperson, List<Sale>>();

public void makeSale(Salesperson salesPerson, Sale sale) {
    List<Sale> sales = map.get(salesPerson);
    if (sales == null) {
        sales = new ArrayList<Sale>();
        map.put(salesPerson, sales);
    }
    sales.add(sale);
}
```



# Multimaps, after

Would you rather do this?

```
Multimap<Salesperson, Sale> multimap  
    = ArrayListMultimap.create();
```

```
public void makeSale(Salesperson salesPerson, Sale sale) {  
    multimap.put(salesPerson, sale);  
}
```

The code on the last slide is

- Verbose
- Bug-prone
- Limited in functionality
- Using the wrong abstraction



# About Multimaps

A collection of key-value pairs (entries), like a Map, except that keys don't have to be unique.

```
{a=1, a=2, b=3, c=4, c=5, c=6}
```

`multimap.get(key)` returns a modifiable Collection *view* of the values associated with that key.

Sometimes you want to think of it as a `Map<K, Collection<V>>`  
-- use the `asMap()` view:

```
{a=[1, 2], b=[3], c=[4, 5, 6]}
```



# Multimap subtypes

- ListMultimap: the get() view implements List
  - preserves the ordering of entries per key; can have duplicate entries
- SetMultimap: the get() view implements Set
  - no duplicate entries, ordering of entries is impl-dependent
- SortedSetMultimap: the get() view implements SortedSet
  - you get the idea

Hmm... sounds a lot like a plain old Map<K, Collection<V>>?

But wait...



# Multimap example 2, before

Now we want to find the biggest Sale. Without Multimap:

```
public Sale getBiggestSale() {
    Sale biggestSale = null;
    for (List<Sale> sales : map.values()) {
        Sale myBiggestSale = Collections.max(sales,
            SALE_CHARGE_COMPARATOR);
        if (biggestSale == null ||
            myBiggestSale.getCharge() > biggestSale().getCharge()) {
            biggestSale = myBiggestSale;
        }
    }
    return biggestSale;
}
```



# Multimap example 2, after

With Multimap:

```
public Sale getBiggestSale() {  
    return Collections.max(multimap.values(),  
        SALE_CHARGE_COMPARATOR);  
}
```

View collections are very powerful. Multimap has six: `get()`, `keys()`, `keySet()`, `values()`, `entries()`, `asMap()`.



# Multimap vs. Map

- Most Map methods are identical on Multimap:
  - `size()`, `isEmpty()`
  - `containsKey()`, `containsValue()`
  - `put()`, `putAll()`
  - `clear()`
  - `values()`
- The others have analogues
  - `get()` returns `Collection<V>` instead of `V`
  - `remove(K)` becomes `remove(K,V)` and `removeAll(K)`
  - `keySet()` becomes `keys()` (well, and `keySet()`)
  - `entrySet()` becomes `entries()`
- And Multimap has a few new things
  - `containsEntry()`, `replaceValues()`

# Multimap implementations

---

- ImmutableMultimap
- ArrayListMultimap
- HashMultimap
- LinkedHashMapMultimap
- TreeMultimap

# Overview

---

1. In a nutshell
2. Immutable Collections
3. Multisets
4. Multimaps
- 5. Other new types/impls**
6. Static utilities
7. Other stuff
8. Q & A

Questions also welcome throughout!



# BiMap

- aka unique-valued map, it guarantees its values are unique, as well as its keys
- Has an `inverse()` view, which is another BiMap
  - `bimap.inverse().inverse() == bimap`
- Stop creating two separate forward and backward Maps!
  - Let us do that for you.

## Implementations:

- `ImmutableBiMap`
- `HashBiMap`
- `EnumBiMap`



# ReferenceMap

- (If you haven't used weak or soft references, take a one-minute nap.)
- A generalization of `java.util.WeakHashMap`
- Nine possible combinations:
  - strong, weak or soft keys
  - strong, weak or soft values
- Fully concurrent
  - implements `ConcurrentMap`
  - cleanup happens in GC thread
- Isn't yet as fast as it could be, but we use it anyway

# Ordering class

Comparator is easy to implement, but a pain to use.  
Ordering is Comparator++ (or RichComparator).

```
Ordering<String> caseless = Ordering.forComparator(  
    String.CASE_INSENSITIVE_ORDER);
```

Now you have tasty methods like `min(Iterable)`, `max(Iterable)`, `isIncreasing(Iterable)`, `sortedCopy(Iterable)`, `reverse()`...



# Overview

---

1. In a nutshell
2. Immutable Collections
3. Multisets
4. Multimaps
5. Other new types/impls
- 6. Static utilities**
7. Other stuff
8. Q & A

Questions also welcome throughout!



# Static factory methods

- We have them.
- Rather than asking you to type

```
Multimap<String, Class<? extends Handler>> handlers =  
    new ArrayListMultimap<String, Class<? extends  
Handler>>();
```

- you type

```
Multimap<String, Class<? extends Handler>> handlers =  
    ArrayListMultimap.create();
```

- We provide these for JDK collections too, in classes called Lists, Sets and Maps.
- With overloads to accept Iterables to copy elements from



# Working with Itera\*s

- Collection is a good abstraction when all your data is in memory
- Sometimes you want to process large amounts of data in a single pass
- Implementing Collection is possible but cumbersome, and won't behave nicely
- Iterator and Iterable are often all you need
- Our methods accept Iterator and Iterable whenever practical
- And ...

# "Iterators" and "Iterables" classes

These classes have parallel APIs, one for Iterator and the other for Iterable.

```
Iterable transform(Iterable, Function) *  
Iterable filter(Iterable, Predicate) *  
T find(Iterable<T>, Predicate)  
Iterable concat(Iterable<Iterable>)  
Iterable cycle(Iterable)  
T getOnlyElement(Iterable<T>)  
Iterable<T> reverse(List<T>)
```

These methods are LAZY!

\*No, this doesn't make Java into an FP language.



# Overview

---

1. In a nutshell
2. Immutable Collections
3. Multisets
4. Multimaps
5. Other new types/impls
6. Static utilities
7. **Other stuff**
8. Q & A

Questions also welcome throughout!



# What we're not telling you about

- Forwarding collections
- Constrained collections
- Precondition checking
- Union/intersection/difference for Sets
- `Multimaps.index()` and `Maps.uniqueIndex()`
- `ClassToInstanceMap`
- `ObjectArrays` utilities
- `AbstractIterator` and `PeekingIterator`
- `Join.join()`
- and more

# How do we test this stuff?

We have over 25,000 unit tests. How?

```
return new SetTestSuiteBuilder()
    .named("Sets.filter")
    .withFeatures(GENERAL_PURPOSE, ALLOWS_NULL_VALUES,
                 KNOWN_ORDER, CollectionSize.ANY)
    .using(new TestStringCollectionGenerator() {
        public Set<String> create(String[] elements) {
            return Sets.filter(...);
        }
    })
    .createTestSuite();
```

- Googlers George van den Driessche and Chris Povirk
- We'll open-source this framework too
- Several JDK collections don't pass all its tests!



A decorative header at the top of the page features four overlapping spheres. From left to right, they are light green, light blue, light red, and light yellow. A thin black horizontal line runs across the page just below the spheres.

# Q & A

<http://google-collections.googlecode.com>

