Idea Graveyard

The goal for this page is to flesh it out to contain a reasonably complete explanation of why we rejected each of these ideas.

Something missing? Search for an issue report, or file a new one. Let us know on the issue discussion if you think an idea belongs here.

com.google.common.base

Tuples for n >= 2

Tuple types are awful obfuscators. Tuples obfuscate what the fields actually mean (`getFirst` and `getSecond` are almost completely meaningless), and anything known about the field values. Tuples obfuscate method signatures: `ListMultimap<Route, Pair<Double, Double>>` is much less readable than `ListMultimap<Route, LatLong>` .

StackOverflow has some explanation here, too.

Instead, we released AutoValue. This makes it easy to just create your own proper value classes.

Functions/Predicates for n >= 2 inputs

Part of the debate here is similar to the debate over tuples: When should we use generic interfaces, and when should we use specialized interfaces? The way we've leaned in Guava is toward specialized interfaces: We provide Range instead of Pair, and we provide Equivalence instead of BinaryPredicate. (If we could convince ourselves to add reduce(), we'd probably provide Reducer instead of BinaryFunction. But more likely still is that we'd provide methods like IntMath.sum.) We even have CharMatcher in addition to Predicate, CacheLoader in addition to Function, and so forth.

Another part of the debate is about widespread functional programming in Java. In a pre-Java 8 world, the language's verbosity (especially for anonymous classes and for generics) gets in our way. We've written about the downsides of functional programming in Java before. Still, there are cases in which it makes sense. We've tried to capture most of those with Function and Predicate, but if you need more complex types, have a look at other Java libraries for functional programming.

Why did we draw the line at single-input functions and predicates? It was a judgment call based on the frequency that single-input and multi-input versions are used, the growing verbosity as the number of inputs increases, and the temptations of overuse. While we can imagine code that needs to perform a SQL JOIN on two Maps, join(Map<K, V1>, Map<K, V2>, BiFunction<V1, V2, O>) is unlikely to be a widely usable solution, for functionality, performance, and verbosity reasons. We would rather add a pseudo-database class (maybe one day...) than provide short wrappers that leave users wanting more.

`Predicates.sameAs()`

See Issue 355 (thanks, lnezda).

We experimented with this internally, but we found that all of its users could be better served in other ways.

`Preconditions.checkArgumentNotNull()` (throws IllegalArgumentException)

We realize there are many valid arguments in favor of throwing IAE on a null argument. In fact, if we had a time machine to go back >15 years, we might even try to push things in that direction. However, we have decided to stick

with the JDK and *Effective Java* preference of `NullPointerException`.

If you're steadfast in your belief that IAE is right, you still have `checkArgument(arg != null)`, just without the convenience of it returning `arg`, or you can create a local utility to your project.

com.google.common.collect

`Iterables.isNullOrEmpty`

Prefer to return empty collections instead of null. Then a plain `isEmpty` check will suffice. (Thanks, xaerxess.)

counting/indexed iterator

A CountingIterator has a variety of possible uses, and I saw most of them in going through our internal Google uses. Roughly from most common to least, they are:

1. CountingIterator was used in our GXP templating system's gxp:loop element. It used to be difficult to count elements in GXP. `gxp:loop` has since been changed a while back to support a `key` attribute. It defines a variable in which the count is kept. This turns out to be much nicer:

Before:

```
<gxp:abbr name='iter' type='CountingIterator{String}'
expr='CountingIterator.from(keys)'>
  <gxp:loop var='value' type='String' iterator='iter'>
    <gxp:abbr name='index' type='int' expr='iter.getIndex()'>
```

After:

```
<gxp:loop var='value' type='String' key='index' iterable='keys'>
```

2. CountingIterator was used in otherwise typical for() loops. Usually, removing CountingIterator simplified the code:

Before:

```
CountingIterator<Record> iterator = CountingIterator.from(records);
while (iterator.hasNext()) {
  Record record = iterator.next();
  int i = iterator.getIndex();
```

After:

```
for (int i = 0; i < records.size(); i++) {
  Record record = records.get(i);
```

Or, if indexed access isn't a good idea:

Before:

```
CountingIterator<Map.Entry<String, Class>> iter =
CountingIterator.from(map.entrySet());
while (iter.hasNext()) {
  doStuff(iter.next(), iter.getIndex());
```

After:

```
Iterator<Map.Entry<String, Class>> iter = map.entrySet().iterator();
for (int i = 0; iter.hasNext(); i++) {
  doStuff(iter.next(), i);
```

3. CountingIterator has shown a particular ability to draw users away from APIs that do what they want more directly. Some people looked for Iterables.limit there -- and found it, since our CountingIterator happened to support it. Others used to use CountingIterator to reimplement partitioning:

Before:

```
CountingIterator<Task> iter = CountingIterator.from(tasks);
while (iter.hasNext()) {
  List<Task> batch = Lists.newArrayList();
  do {
    batch.add(iter.next());
  } while (iter.hasNext() && (iter.getCount() % TASKS_PER_BATCH) > 0);
  // Plus, the above test could have used batch.size() and
  // not needed CountingIterator.
```

After:

```
for (List<Task> batch : partition(tasks, TASKS_PER_BATCH)) {
```

4. CountingIterator can be used to test that an Iterator's contents are lazily evaluated. This works well enough, though there's also some uncertainty around what to track: (a) only the number of calls to next() or (b) any call that looks ahead at a new element (i.e., the number of next() calls, plus 1 if there was a trailing hasNext() call). As an alternative, you may be able to use a ListIterator, which exposes nextIndex().

5. CountingIterator can be defined as an `Iterator<Counted<E>>` so that it can be used as input to methods like transform(), filter(), and toMap(). However, such code tends to be unusually verbose as Java functional code goes. It's a candidate for rewriting with a for() loop.

One more thing that I discovered was some weirdness in how CountingIterator's getIndex() method behaves. The method's value is incremented at each call to next(), so at any given time, getIndex() may be equal to the "natural" loop index (after a call to next()) or one greater (before the call to next() -- notably including after the end of the loop, at which point it's equal to size()). Statefulness is a necessary property of iterators, but it gets worse here: You need to call either next() or getIndex() first, and your choice affects the value in subtle ways. The `Iterator<Counted<E>>` approach outlined above can solve the statefulness problem, though it makes CountingIterator less suitable for the lazy-evaluation testing described in (4).

Given the usages we found, we thought it best not to include a CountingIterator. The class is straightforward to implement atop ForwardingIterator when needed.

Lazy/computing Map whose get() method returns a default value

Our main strategy for satisfying `LazyMap` use cases has been to add specialized collection types: `LazyMap<K, List<V>>` becomes `Multimap<K, V>`, `LazyMap<K, Integer>` becomes `Multiset<K>` or `AtomicLongMap<K>`, `LazyMap<K1, Map<K2, V>>` becomes `Table<K1, K2, V>`, some `Maps` become `LoadingCaches`. That is in part because those abstractions are more powerful. It's also in part because `LazyMap` has surprising behavior in some cases. For example, `lazyMap.equals(myMap)` is likely to behave as expected, but `myMap.equals(lazyMap)` can cause entries to be inserted into `lazyMap`. A mutating `equals()` is pretty weird on its own, and it only gets weirder when it's so easy to change a non-mutating call to a mutating one (say, if someone rewrites `HashMap.get` to call `a.equals(b)` instead of `b.equals(a)`).

Given that, I suggest first looking at our various collection types. If that won't work, try wrapper methods with implementations containing snippets like `firstNonNull(map.get(key), defaultValue)` or `Functions.fromMap(map, defaultValue).apply(key)`. Changing `get()` itself, though, produces strange enough behavior that we've removed the feature from Guava (which used to support it).

A method to view an iterator as an iterable

The biggest concern is that `Iterable` is generally assumed to be able to produce multiple independent iterators. The doc doesn't say this, but the `Collection` doc doesn't say this, either, and yet we assume it of its iterators. We have had breakages in Google when this assumption was violated.

The simplest workaround is `ImmutableList.copyOf(Iterator)`, which is pretty fast, safe, and provides many other advantages besides.

`Lists.filter`

See issue 505.

The biggest concern here is that too many operations become expensive, linear-time propositions. If you want to filter a list and get a *list* back, and not just a `Collection` or an `Iterable`, you can use `ImmutableList.copyOf(Iterables.filter(list, predicate))`, which "states up front" what it's doing and how expensive it is.

more varargs factory methods (e.g. `Lists.newLinkedList(E...)`)

Almost all the cases we've encountered for this in reality would really have been better off with an immutable collection; they never actually changed the collection.

For the remaining cases, using a copy factory wrapped around `Arrays.asList` is a perfectly good one-line workaround. Alternatively, you *are* building a *mutable* collection, so you can afford to use e.g. `Collections.addAll(Collection<E>, E...)`.

create a map from an `Iterable<Pair>`, `Object[]` (alternating keys and values), or from `List<K>` + `List<V>`

TODO(kevinb): fill in

Note that we have added `ImmutableMap.copyOf(Iterable<Entry>)`.

`Lists.equalIgnoringOrder(list1, list2)`

As Kevin stated on StackOverflow, "the fact that you want to do this in the first place is a strong signal that you want one or both of these collections to be represented as Multisets in the first place. They cannot really logically be Lists if you don't care about their order. If you do represent both as Multisets, then guess what? You've got your single-method-call solution!"

The single-line solution for any arbitrary pair of `Iterables`, by the way, is `return ImmutableMultiset.copyOf(elems1).equals(ImmutableMultiset.copyOf(elems2));`

which is linear-time, linear-memory, and makes those costs obvious up front.

`Iterables.countMatching(Iterable, Predicate)`

A fully equivalent solution is `return Iterables.size(Iterables.filter(iterable, predicate));`

`Sets.transform()`

See issue 219.

A `Set` with a slow `contains` method is a non-starter, and a `Function` isn't necessarily bijective. Prefer `ImmutableSet.copyOf(Collections2.transform(set, func));`

Persistent mutable collections/mutation methods on immutable collections

Wikipedia defines a persistent data structure as one which "always preserves the previous version of itself when it is modified." Efficient persistent data structures are a well-known area of research, and many sophisticated structures are known.

Guava's immutable collections, on the other hand, are *not* designed as persistent data structures. They are designed to implement -- extremely efficiently, both in terms of CPU time and memory -- collections which do not change, in any way, after being built, and they aim to be at least as efficient as the corresponding mutable collection types. Rewriting these data structures for persistence would entail significant constant-factor (or worse) overhead to the overwhelming majority of users of these types.

While it might be possible to provide a parallel API to support *inefficiently* modifying Guava immutable collections -- e.g. a method on `ImmutableList` to create a copied `ImmutableList` with a new element added -- we feel this is only likely to confuse users who expect those methods to have sublinear performance, and assume we have implemented efficient persistent data structures.

Alternatives include using a traditional mutable collection, explicitly creating a new immutable collection with a builder, or using another Java library specifically intended to provide persistent data structures.