TOOLS

Guava: A treasure trove of Java functionality

This broad collection of utilities and functions from Google merits a principal place in your toolkit.

by Andrew Binstock

April 30, 2021

Guava is an omnibus open source Java library created by Google for its internal needs. Within this library, you'll find a treasure chest of functionality, some of which I'll discuss in this article—and all of which is actively maintained by Google. Due to Guava's wide adoption within the company and the larger community, the library has grown steadily: The current version is its 30th major release.

Omnibus libraries such as Guava are libraries that cover a wide range of functionality. They are comparatively rare these days, because developers have come to prefer cobbling together lots of single-purpose libraries via complex builds. The only remaining omnibus solutions of any kind in Java are frameworks. But as you'll see, a library such as Guava can do away with many dependencies and greatly simplify your projects.

I'll explore the range of functionality shortly, but first it's important to point out that some of that functionality duplicates capabilities introduced in Java releases—in the language as well as the core libraries—after Java 8. This is intentional, because Guava runs on Java 8 and provides older JDK users the ability to use features similar to those found in later Java releases. With this in mind, there are two separate versions of Guava, the Java 8 version, which I discuss in this article, and the Android version, which is written for the language equivalent of Java 1.6.

To include Guava in your project, add this entry to your Maven file.

```
<dependency>
  <groupId>com.google.guava</groupId>
  <artifactId>guava</artifactId>
  <version>30.1.1-jre</version>
</dependency>
```

Users of Gradle should consult the Guava homepage for the multiplicity of options for adding Guava to their projects.

Overview of the functionality

I first came across Guava in the days of Java 6. The library offered a small option that appealed to my desire for security in the correct operation of my code: preconditions. These methods, which are still in Guava, let you check the preconditions in a method prior to calling a given method. For example, you can check that a passed argument is not null, as follows:

```
Preconditions.checkNotNull(Object, message))
```

This call returns the object if it's not null. So, when you write the code that assigns arguments to local variables, you can run them through this check first. If they're null, a null pointer exception is thrown with your specified message.

Guava offers other similar precondition tests, such as `checkArgument()` and `checkPositionIndex()`. The latter call, by default, checks for a value between 0

and the maximum size of the given array. What appealed to me about these functions was that they replaced the more troublesome assertion mechanism. Assertions work fine, except that they must be specifically enabled (using `-ea` on the command line); otherwise, they are disabled. I wanted assertions that would *always* be enabled without having to impose any burden on the user to set up the command line correctly.

Java SE ultimately added similar capability. In Java 7, the `Objects` class provided a similar set of checks as static methods. In Java 9, the number of test methods was increased significantly. But by then, I had begun doing what you do with any omnibus library: using many of its other features.

The Wikipedia article on Guava does a good job of summarizing those features as

> basic utilities to reduce manual labor to implement common methods and behaviors; an extension to the Java collections formerly called the Google Collections Library; and other utilities that provide convenient and productive features such as functional programming, graphs, caching, range objects, and hashing.

Here are some of my favorite goodies.

Collections

As mentioned in the Wikipedia article, Guava grew out of Google Collections, so the set of collections it provides is deep indeed. Let's begin with immutable collections, which are a particularly valuable and somewhat underappreciated set of resources.

Since the early days of Java's `Collections` class, you could use unmodifiable options, such as`Collections.unmodifiableSet` and its siblings: list and map. However, it wasn't until Java 9 that you could create these data structures using fluid method calls such as `Map.of`, and it was not until Java 10 that you could enjoy `Map.copyOf`. Guava has offered this kind of functionality for many releases, and so if you're forced to use releases earlier than Java 10, Guava gives you a solution—and a lot more.

For example, Guava provides the methods I've just mentioned but also implements the Builder pattern, so you can create a collection from multiple sources.

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

Guava provides immutable variants of all the standard collections (set, list, table, and so on), and it adds some additional, very handy data structures.

The first of these is the *BiMap*, or bidirectional map. A BiMap solves a common problem that occurs with standard maps that are built on conventional key-value pairs: Sometimes you need the values to be keys and the keys to be values. With the BiMap, each entry serves as a key-value pair that can be reversed, so you can use the value to point to a key. The only requirement is that all keys and values be unique (within each group, rather than within the entire map).

A related and equally useful data structure in Guava is the *multimap*. This construct solves a problem that occurs routinely in Java programming: a key-value map in which the value is a collection of some kind, such as a list. I used a structure of this kind in my article on locating duplicate files in this magazine. The principal table was keyed on file size, and the values consisted of a list of one or more files that were of that size. That multimap looked something like **Figure 1**.

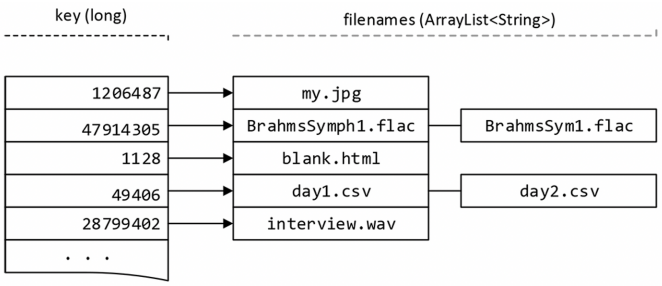

Figure 1. A multimap

While this was a straightforward implementation, multimaps can become very complex and therefore error-prone, and the standard Java library has no support specifically for multimaps. The Guava implementation is interesting in that `Multimap` is an interface and comes with several builders that enable fluid construction. For example, if you wanted to build a multimap where the keys were held in a tree and each value was an `arrayList`, you can do so with the following code:

```
ListMultimap<String, Integer> treeListMultimap =
    MultimapBuilder.treeKeys().arrayListValues().build();
```

Adding entries requires none of the walking of the respective data structures but is a simple call, `put( K, V)`; likewise with deletion and then the handy, if rarely used, `replace()`, which deletes the given collection of values and starts a new one for the specified key. As you'd expect, Guava includes multimaps for several combinations of data structures and, of course, immutable variants of those.

While there are several more interesting and innovative data structures, I'd be remiss not to discuss Guava's *tables*, which are conceptually similar to databases but much simpler in construction and operation. An entry in a table consists of a series of data items, such as first name, last name, street address, city, state, postal code, and country. Every field is treated and indexed as part of a column. In this way, you can read entries as records or you can read (and select) one or more columns—that is, on any field in the table.

Tables can be helpful with graph data and also in situations in which a key-value lookup returns a data item that will then be used as a key in another map, which leads to a key in yet another map. With tables, that kind of chained lookup is replaced by a single entry with all the data fields. Guava offers several versions of tables, including, of course, an immutable implementation.

Lightweight cache

Guava provides several lightweight versions of features typically used in enterprise applications, including a cache and a publish-subscribe (pub-sub) solution.

The Guava cache is lightweight in the sense that it is local exclusively to your currently executing application. However, feature-wise it has many of the capabilities you'd expect in a cache. For example, you can populate the cache through the addition of individual entries or through a group loader.

The cache works as a classic key-value store and cache values are returned if they are present; otherwise, they are computed via callable methods and inserted into the cache. You can configure the eviction method to your preference. Options include eviction based on size, on the absence of references, or on timing. Left to its own devices, the cache evicts on a least-recently used (LRU) basis. In addition, of course, you can manually evict individual entries.

Unlike other caches, the Guava cache does not use a thread in the background constantly monitoring entries looking for which ones to clean up. It does that maintenance work only on insertions and on reads. This means that, for example, items can remain in the cache even after their allotted time has expired or when there are no other references to them. Eventually, if there is memory pressure on the cache, those items will be collected and removed. However, if you need a different eviction scheme, you can use the Guava cache API to add your own preferred eviction mechanism.

A final bit of sophistication: You can collect statistics on the operation of the cache via `CacheBuilder.recordStats()`, which provides data on hit rate, eviction count, and the amount of time spent loading entries.

This data enables you to test and fine-tune the cache loading and eviction configurations for your application.

For situations that require a more robust caching solution, the Guava authors recommend Caffeine, which is a standalone cache that uses the same APIs as the Guava cache. This is due, in part, to the fact that the author of Caffeine, Ben Manes, worked on the Guava cache. Caffeine is also open source and it provides a natural graduation path that facilitates the adoption of Guava early in a project's deployment when demands are light.

Publish-subscribe with EventsBu

Guava's EventBus is an intraprocess pub-sub facility. To quote its documentation, EventBus enables

> publish-subscribe-style communication between components without requiring the components to explicitly register with one another (and thus be aware of each other). It is designed exclusively to replace traditional Java in-process event distribution using explicit registration. It is not a general-purpose publish-subscribe system, nor is it intended for interprocess communication.

The EventBus has several kinds of uses: The principal one would be for communicating events that come from multiple external sources. It can also be used advantageously for what aspect-oriented programming calls *cross-cutting concerns*, such as logging. For example, a loosely coupled modular application still needs a centralized logging function, which could be implemented by Guava's EventBus. An example of such an application would be a Java/Jakarta EE application that makes use of Enterprise JavaBeans.

At its heart, EventBus is an event queue, but the usual Java tasks of setting up listeners, performing notifications, and capturing and distributing events have all been abstracted away to a simple model that has been tested extensively by Google.

The Guava library does not stop there. It also offers deep support for graph modeling, that is, a library for modeling graph-structured data—namely entities and the relationships between them. While I am not sufficiently expert in graph models to communicate the details, colleagues I spoke with for this article expressed great admiration for Guava's graph capabilities. A look at the Wiki page for graphs provides a detailed overview.

In addition to the items I've just discussed, Guava offers a variety of handy, low-level routines. These include a set of hashing functions (both hash table–oriented and cryptographic), string functions (particularly for complex joining, splitting, and handling of tricky character sets), parsing and validating URLs, and so forth.

Extensions to Java primitive data types

One of the longtime frustrations with Java's primitive data types is that there are no unsigned integers. Whereas most languages not named JavaScript or Dart provide unsigned integers, Java has never done so. Guava provides unsigned bytes, integers, and longs. It also provides boxed wrappers for these data types and thoughtfully includes the basic functions you'd expect in support of these types: minimum, maximum, and comparison functions; conversion to `BigDecimal`; and, of course, `toString()`.

The library also offers a way to handle big-endian values. While big-endian values are now rare (with the demise of RISC chips, in general), when they do arise, handling those values can require care, which is largely solved by Guava's functions.

Conclusion

Guava is a seductive library: You begin using it for one thing and soon it's in use throughout your applications doing all kinds of heavy lifting, while adding only a single dependency to your build. In addition to this convenience, the quality of the implementation is high, given that the library is used daily throughout Google. So, Guava has been battle-tested.

When Guava first came out, it was widely picked up by Java developers, who soon became frustrated with its frequent breaking changes that compelled them, at times, to have two instances of the library in their application to handle the version skew. Fortunately, this has not been a problem for several years now: The APIs are stable and mostly expanded rather than abruptly modified.

My experience with this stability is consistent with a recent post by Kevin Bourrillion, the lead developer of Guava: "We were too loose about breaking changes, and I can only apologize for all the trouble that's caused. I'll take that blame personally. I think you will see that our breaking-changes policy got much stricter a couple years ago."

As is clear from the examples I presented earlier, Guava has been good at offering functionality that is later incorporated into standard Java. For this reason, Guava is especially apt for organizations that want current Java features but are constrained to work on Java 8. For all developers, however, the wide range of functionality, particularly the high-level utilities (EventBus, lightweight cache, graphing, and so on) and the collections, make it a rich and useful toolkit.

Dig deeper

- Guava on GitHub
- Designing and implementing a library
- Working and unit testing with temporary files in Java
- Containerizing apps with jlink

Andrew Binstock

Andrew Binstock (@platypusguy) was formerly the editor in chief of *Java Magazine*. Previously, he was the editor of *Dr. Dobb's Journal*. He co-founded the company behind the open-source iText PDF library, which was acquired in 2015. His book on algorithm implementation in C went through 16 printings before joining the long tail. Previously, he was the editor in chief of *UNIX Review* and, earlier, the founding editor of the *C Gazette*. He lives in Silicon Valley with his wife. When not coding or editing, he studies piano.

Share this Page