Guava's Philosophy

In progress

What Guava Is

Guava is the open-sourced version of Google's core Java libraries: the core utilities that Googlers use every day in their code. The Guava utilities have been carefully designed, tested, optimized and used in production at Google. You don't need to write them, test them, or optimize them: you can just use them.

Guava is a *productivity multiplier* for Java projects across the board: we aim to make working in the Java language more pleasant and more productive. The JDK utilities, e.g. the Collections API, have been widely adopted and have significantly simplified virtually all Java code. We hope to continue in that tradition.

I Could've Invented That

Effective Java item 47, "Know and use the libraries," is our favorite explanation of why using libraries is, by and large, preferable to writing your own utilities. The final paragraph bears repeating:

> *To summarize, don't reinvent the wheel. If you need to do something that seems like it should be reasonably common, there may already be a class in the libraries that does what you want. If there is, use it; if you don't know, check. Generally speaking, library code is likely to be better than code that you'd write yourself and is likely to improve over time. This is no reflection on your abilities as a programmer. Economies of scale dictate that library code receives far more attention than most developers could afford to devote to the same functionality.*

We'd also like to mention that:

- Guava has been battle-tested in production at Google.
- Guava has staggering numbers of unit tests: as of July 2012, the guava-tests package includes over 286,000 individual test cases. Most of these are automatically generated, not written by hand, but Guava's test coverage is *extremely* thorough, especially for `com.google.common.collect`.
- Guava is under active development and has a strong, vocal, and involved user base.
- The best libraries seem obvious in retrospect, but achieving this state is notoriously challenging.

When In Doubt...

We can be somewhat conservative on adding features to Guava. Guava sits so low in the stack that removing features without breaking users is extremely difficult, so we tend to hold off on adding features we're not sure about. (Sometimes, we'll trial-run features "Google-internally," where we can migrate users away from a failed feature ourselves.)

Guava's primary metric when deciding whether to add a new feature is frequently summed up as *utility times ubiquity*.

A feature is said to have *utility* when it represents a substantial improvement on the simplest available workaround. In principle, there is always *some* workaround -- even if it's just writing that utility method, or that data structure, yourself. On the other hand, adding a feature to Guava might

- save you significant amounts of code
- avoid forcing you to write code that's difficult to debug, or that's easy to get wrong
- improve readability
- improve speed

as compared to the available workarounds.

A feature has *ubiquity* when there are a diverse range of use cases for it.

When trying to estimate the ubiquity of a feature, we frequently use the Google internal code base as a reference: if not one project inside Google found a use for some code, then does *anybody* really need that feature? Having such a large code base to examine gives us lots of data to work with...and it's always nice to be able to cite hard numbers.

Not all Guava features have much utility (see e.g. `Lists.newArrayList`) or ubiquity (see e.g. `MinMaxPriorityQueue`), but we generally try to maximize their product.

A few general design principles Guava aims for are:

- There should always be some use case for which the API is clearly the *best* solution possible. If we're not sure this is the best API possible for the job, think about it until we are sure.
- The semantics of methods and classes should be obvious and intuitive from their signatures, as opposed to "smart." Internal implementations can do smart things in special cases, but the semantics for those special cases should be the same.
- Encourage good code habits in Guava users, and exemplify good code habits ourselves in Guava source. (This has many corollaries, including "fail fast," "reject nulls," and the like.)
- Don't try to address every use case individually -- provide generic tools that can be composed to address use cases we haven't thought of yet.
- Emphasize maintainability, and leave room for future refactorings. (Corollary: most exposed classes should be `final`, as per Effective Java item 17; exposing "skeleton" `AbstractXXX` classes should be done very conservatively.)

Iteration

In contrast to the JDK -- which maintains extremely strict backwards compatibility even for its worst mistakes -- Guava deprecates, and yes, deletes unwanted features over time. It is important to us that when you see a feature in the Javadocs, it represents the Guava team's best work, and not a feature that in retrospect was a bad idea.

We do, however, work to make sure that we don't wantonly break our users. If we aren't ready to freeze the API on some particular component, for whatever reason, we will mark the component as @Beta. More details on this are below.

Beta APIs

Beta APIs represent Guava features that we aren't ready to freeze for *whatever* reason: because the methods might not find enough users, because they might be moved, because their uses might be too narrow to include them in Guava.

That said, @Beta APIs are fully tested and supported, and treated with all the care and affection that the rest of Guava receives.

The biggest connotation of the @Beta annotation is that annotated classes or methods are subject to change. They can be modified in any way, or even removed, at any time. If your code is a library itself (i.e. it is used on the CLASSPATH of users outside your own control), you should not use beta APIs, unless you repackage them (e.g. using ProGuard).

All this said, @Beta features *tend* to remain relatively stable. If we decide to delete a @Beta feature, we will typically deprecate it for one release before deleting it.

On the other hand, if you want something taken out of @Beta, *file an issue*. We generally promote features out of @Beta only when it's specifically requested, so if you don't ask, it won't happen.

Non-Beta APIs

APIs without `@Beta` will remain binary-compatible for the indefinite future. (Previously, we sometimes removed such APIs after a deprecation period. The last release to remove non-`@Beta` APIs was Guava 21.0.) Even `@Deprecated` APIs will remain (again, unless they are `@Beta`). We have no plans to start removing things again, but officially, we're leaving our options open in case of surprises (like, say, a serious security problem).

Support and Feedback

The Guava community is very active. Generally speaking:

- Ask for help on a specific question or problem on StackOverflow.
- File feature requests, bug reports, and the like under Issues.
- General discussion that doesn't fit into these categories takes place on our discussion group.

The Guava team is generally highly active in all of these areas.