Compatibility

Android

Starting with v21.0, Guava requires JDK 8. Starting with v22.0, Guava has an Android flavor as well, which you can use by appending `-android` to the version string when building.

It's especially important when using Guava in Android applications to use ProGuard to strip out the parts of Guava that aren't used. Guava is one of the most common dependencies for Android applications.

Older JDKs

For the medium-term future, Guava users who need to target JDK 7 can use the Android flavor, which has no Android-specific dependencies. At some point in the future, when Android diverges sufficiently from JDK 7, Guava may stop providing a JDK 7-compatible flavor, at which time those users will have to stay with the latest prior version.

Guava users who need to target JDK 6 will have to stay on v20.0.

GWT

Much of Guava is compatible with Google Web Toolkit (GWT), though we do not currently optimize code specifically for GWT. Guava APIs compatible with GWT are marked with the annotation `@GwtCompatible`.

Backward compatibility

Our README file covers the basics: The only APIs we expect to remove are `@Beta` methods and classes.

"Source-compatible since Guava release xx"

Sometimes we will add a more specific overload of an existing method. Code compiled against the old version of Guava will still run against the new version of Guava. Additionally, any code that compiles against the new version of Guava will also compile against the old version of Guava. However, code compiled against the new version of Guava might not *run* against the old version because it may compile against the new overload.

An example of this is `MoreObjects.toStringHelper`, which originally accepted an Object but which now has `String` and `Class` overloads, as well. `Object.toStringHelper(anything)` compiles under any version of Guava, but if the static type of `anything` is `String` or `Class`, a compile against a recent Guava version will select a new overload.

"Mostly source-compatible since Guava release xx"

As of this writing, there are two causes of this:

First, we sometimes change a `@Beta` interface to a class. Simple anonymous uses like `new Foo() { ... }` will continue to compile, but named classes will not. In neither case is the result binary compatible.

Second, we sometimes replace a `@Beta` method with a version that returns a more specific type. In this case, code compiled against a version of Guava from before the change might not run against a version of Guava from after the change, and vice versa. However, code that compiles against the old version of Guava will *almost* always compile against the new version of Guava. (The reverse is often true, too, when the code doesn't take advantage of the new return type.)

However, there are rare exceptions. Consider this fictional case:

Guava release n:

```
public static List<Integer> newList() { ... }
```

Guava release n + 1:

```
public static ArrayList<Integer> newList() { ... }
```

Most callers will compile fine against either version:

```
List<Integer> myList = newList();
```

Of course, if a caller uses the new, specific type, that code won't compile against the old version:

```
ArrayList<Integer> myList = newList();
```

The more interesting case, though, is code that compiles against the old version but not the new version:

```
Set<List<Integer>> myLists = ImmutableSet.of(newList());
```

Java's type inference isn't strong enough for `ImmutableSet.of()` to realize that it should treat its input as a `List<Integer>` instead of an `ArrayList<Integer>`, so the code must be rewritten:

```
Set<ArrayList<Integer>> myLists = ImmutableSet.of(newList());
```

Or, to produce code that compiles against either version of Guava:

```
Set<List<Integer>> myLists = ImmutableSet.<List<Integer>>of(newList());
```

Or, equivalently:

```
List<Integer> myList = newList();
Set<List<Integer>> myLists = ImmutableSet.of(myList);
```