Why we deprecated `Throwables.propagate`

When we deprecate a method, the replacement is often an equivalent method with a new name, or at least it's usually still a one-liner. That's not the case for `Throwables.propagate`. Its literal replacement is:

```
throwIfUnchecked(e);
throw new RuntimeException(e);
```

Inlining that would be fine if `propagate` were rarely used, but it's not. Internal to Google, for example, it had over 10,000 callers.

Even if the method is only slightly useful, it is used often enough for that to add up. So why did we deprecate it? We found that its average utility in practice was actually slightly *negative* -- and it's used often enough for that to add up.

Negatives

Can be replaced with a one-liner

We found that 75% of `propagate` calls passed an argument that was already known to be a checked exception...

```
try {
  return something(...);
} catch (IOException e) {
  throw Throwables.propagate(e);
}
```

...which is equivalent to...

```
try {
  return something(...);
} catch (IOException e) {
  throw new RuntimeException(e);
}
```

...or an argument that was already known to be an unchecked exception...

```
try {
  return something(...);
} catch (CancellationException e) {
  throw Throwables.propagate(e);
}
```

...which is equivalent to...

```
try {
  return something(...);
} catch (CancellationException e) {
  throw e;
}
```

...and which, in this particular case, we can simplify further. (See the next section.)

That's not the end of the world, but it means that a majority of calls have no positive value.

Additionally, many of the remaining 25% can get by with `throw new RuntimeException(e)`, too. The downside: If `e` is a `RuntimeException`, the call will unnecessarily wrap it. But if it's acceptable to wrap a checked exception into a generic `RuntimeException`, then it is likely acceptable to do the same for an unchecked exception. (The main case in which wrapping would be undesirable is if there are many nested levels of code that catch `Exception` and `propagate` the result.) So some of the remaining 25% of users are getting minimal value, too.

And the second example above hints at how the value becomes negative:

Obscures no-op `try` - `catch`

Once we simplified code to...

```
try {
  return something(...);
} catch (CancellationException e) {
  throw e;
}
```

...it became obvious that there was no need for the `try` - `catch` at all, giving us...

```
return something(...);
```

Outsmarts reachability detector

We recommended using `propagate` as...

```
throw Throwables.propagate(e);
```

This way, the compiler knows that execution will never continue after the `propagate` statement. A lot of people did this, but many did not. We found code like this...

```
Throwables.propagate(e);
return new MyType<Something>(
    ImmutableList.<Foo>of(),
    ImmutableList.<Bar>of(),
    Something.empty()); // unreachable
```

The problem can be solved by using `throw`, but the problem exists in the first place only because of `propagate`.

(Another problem with `propagate` and the reachability detector: Sometimes users need to add dummy initial values to variables. This is again fixable by using `throw`, but users are unlikely to go back and remove the dummy initial values.)

Obscures in general

During our internal migration, we came across code like this. Both the reviewer and I misinterpreted it, believing that our automated refactoring had done the wrong thing.

```
  try {
    return callable.call();
```

```
  } catch (AssertionError e) {
    int delay = retryStrategy.getDelayMillis(tries);
    if (delay >= 0) {
      try {
        Thread.sleep(delay);
      } catch (InterruptedException ie) {
        throw Throwables.propagate(e);
      }
    }
  }
```

The refactoring changed the `propagate` call to `throw e`. This looked like a bug to us: The `catch` block catches `InterruptedException`, a checked exception, so we should be wrapping it. But we were wrong: `propagate` uses `e`, a parameter from a different `catch` block, not `ie`, the `InterruptedException`.

As usual with confusing code, we can place blame on more than one cause -- including on *us*, as the code looks obvious to me 4 years later (though that might be because I've simplified the code above). But this kind of confusion is inevitable when we write `Throwables.propagate(e)` instead of `e` thousands of times.

`propagate` is magic

We saw code like this:

```
try {
  return something(...);
} catch (SomeException e) {
  Throwables.propagate(e);
  log.log(SEVERE, "error", e);
  return default;
}
```

Someone -- and some reviewer -- thought that `propagate` did something other than throw. Of course those people are wrong, and every API permits misuse. But it's easy enough to see the reasoning here: Why would Guava have a method that's roughly equivalent to the `throw` keyword? And if it did, why would it be named "propagate" instead of something with "throw" in the name? (We could help here by renaming the method, but that causes about as much churn as deleting it.)

`propagate` is magic, part 2

We saw code like this:

```
private Something foo() throws IOException {
  try {
    return something(...);
  } catch (IOException e) {
    log.log(SEVERE, "error", e);
    Throwables.propagate(e);
  }
}
```

It seems pretty likely here that the author expected for `propagate(e)` to rethrow `IOException e` directly. Maybe that person assumed that `propagate` looked like this:

```
<X extends Throwable> void propagate(X x) throws X;
```

Or maybe that person was just new to Java and didn't think about whether this was even possible.

Or maybe the person really meant the code as it is and just forgot to remove `throws IOException`. Or maybe not, but the reviewer assumed that that's what happened (and figured it wasn't worth correcting). We don't know because `propagate` obscures the author's intention.

Encourages unwrapping exceptions from other threads

One common use of `Throwables.propagate` was to unwrap `ExecutionException`:

```
try {
  return cache.get(value);
} catch (ExecutionException e) {
  throw Throwables.propagate(e.getCause());
}
```

This can produce confusing stack traces. Suppose that one request triggers a cache load, and another request reuses it. If the cache load fails with an unchecked exception, the second request fails with a stack trace from the first.

Of course this is not specific to `propagate`, but `propagate` makes it easy. We encourage such users to migrate to `Futures.getUnchecked` and `LoadingCache.getUnchecked`, which are designed for exactly this purpose.

(We've found that `propagate` callers often demonstrate a need for higher-level libraries like this. Another example is `Uninterruptibles`.)

multicatch is now available, and it's safer

Sometimes people used `propagate` to avoid code duplication. They'd change this...

```
public void run() {
  try {
    delegate.run();
  } catch (RuntimeException e) {
    failures.increment();
    throw e;
  } catch (Error e) {
    failures.increment();
    throw e;
  }
}
```

...to this...

```
public void run() {
  try {
    delegate.run();
  } catch (Throwable e) {
    failures.increment();
    throw Throwables.propagate(e);
  }
}
```

Nowdays, they can use multicatch and plain `throw`:

```
} catch (RuntimeException | Error e) {
```

This has the advantage that you can't accidentally catch other kinds of exceptions, as you could with `catch (Throwable e)` + `propagate`.

Confusion about the return type

Since `propagate` is declared to return `RuntimeException`, a few people assumed that it did. We found code like this...

```
logAndRethrow(Throwables.propagate(e))
```

...which will never call `logAndRethrow`.

More commonly (but still in only ~1% of callers), we saw people define helper methods that called `Throwables.propagate` and, like it, declared a return value of `RuntimeException`. This pattern was questionable enough with `Throwables.propagate` itself: Even though the method is well known, at least a few users were confused, as noted above in this section. It's even more dangerous with the less known, undocumented, poorly named helper methods we saw, like `logAndReturn`.

Encourages "not caring about failures"

This point is both squishy and controversial. But it became clear that `propagate` had become something that many people do without thinking about it. That works out OK for many users, but it's not something we like to encourage with a library as widely used as Guava.

Throwing `RuntimeException` makes detecting other bugs harder

Internally, we recommend treating `Throwable`, `Exception`, `Error`, and `RuntimeException` as if they were abstract, and encourage developers to avoid constructing or throwing a new instance of any of these types directly. `Throwables.propagate()` obviously violates this advice.

Throwing `RuntimeException` or these other "top-level" types leaves your callers unable to safely handle your exception without risking catching other unrelated exceptions (like `NullPointerException`) at the same time. Even if you think no one should ever want to catch your exceptions and you just want to fail, there's no need to throw one of these types. Consider `AssertionError` or `VerifyException` in such cases, or another unchecked exception. See the [[Conditional Failures|ConditionalFailuresExplained]] page for more options.

What to do

- You can inline the method, of course.
- In many cases, you'll be able to use `throw e` or `throw new RuntimeException(e)` with no change in behavior. This sometimes lets you remove other code.
- In other cases, you can use `throw new VerifyException(e)` with an inconsequential change in behavior. This is generally preferable to `throw new RuntimeException(e)`.
- In still other cases, you may want to look at the higher-level APIs mentioned above.
- You can create your own copy of the method and migrate to that.