Using and avoiding null

> *"Null sucks."* -Doug Lea
>
> *"I call it my billion-dollar mistake."* - Sir C. A. R. Hoare, on his invention of the null reference

Careless use of `null` can cause a staggering variety of bugs. Studying the Google code base, we found that something like 95% of collections weren't supposed to have any null values in them, and having those fail fast rather than silently accept `null` would have been helpful to developers.

Additionally, `null` is unpleasantly ambiguous. It's rarely obvious what a `null` return value is supposed to mean – for example, `Map.get(key)` can return `null` either because the value in the map is null, or the value is not in the map. Null can mean failure, can mean success, can mean almost anything. Using something other than `null` makes your meaning clear.

That said, there are times when `null` is the right and correct thing to use. `null` is cheap, in terms of memory and speed, and it's unavoidable in object arrays. But in application code, as opposed to libraries, it is a major source of confusion, difficult and weird bugs, and unpleasant ambiguities – e.g. when `Map.get` returns null, it can mean the value was absent, or the value was present and null. Most critically, null gives no indication what a null value means.

For these reasons, many of Guava's utilities are designed to fail fast in the presence of null rather than allow nulls to be used, so long as there is a null-friendly workaround available. Additionally, Guava provides a number of facilities both to make using `null` easier, when you must, and to help you avoid using `null`.

Specific Cases

If you're trying to use `null` values in a `Set` or as a key in a `Map` – don't; it's clearer (less surprising) if you explicitly special-case `null` during lookup operations.

If you want to use `null` as a value in a Map – leave out that entry; keep a separate `Set` of non-null keys (or null keys). It's very easy to mix up the cases where a `Map` contains an entry for a key, with value `null`, and the case where the `Map` has no entry for a key. It's much better just to keep such keys separate, and to think about what it *means* to your application when the value associated with a key is `null`.

If you're using nulls in a `List` – if the list is sparse, might you rather use a `Map<Integer, E>`? This might actually be more efficient, and could potentially actually match your application's needs more accurately.

Consider if there is a natural "null object" that can be used. There isn't always. But sometimes. For example, if it's an enum, add a constant to mean whatever you're expecting null to mean here. For example, `java.math.RoundingMode` has

an `UNNECESSARY` value to indicate "do no rounding, and throw an exception if rounding would be necessary."

If you really need null values, and you're having problems with a null-hostile collection implementations, use a different implementation. For example, use `Collections.unmodifiableList(Lists.newArrayList())` instead of `ImmutableList`.

Optional

Many of the cases where programmers use `null` is to indicate some sort of absence: perhaps where there might have been a value, there is none, or one could not be found. For example, `Map.get` returns `null` when no value is found for a key.

`Optional<T>` is a way of replacing a nullable `T` reference with a non-null value. An `Optional` may either contain a non-null `T` reference (in which case we say the reference is "present"), or it may contain nothing (in which case we say the reference is "absent"). It is never said to "contain null."

```
Optional<Integer> possible = Optional.of(5);
possible.isPresent(); // returns true
possible.get(); // returns 5
```

`Optional` is **not** intended as a direct analogue of any existing "option" or "maybe" construct from other programming environments, though it may bear some similarities.

We list some of the most common `Optional` operations here.

Making an Optional

Each of these are static methods on `Optional`.

Method	Description
`Optional.of(T)`	Make an Optional containing the given non-null value, or fail fast on null.
`Optional.absent()`	Return an absent Optional of some type.

Method	Description
`Optional.fromNullable(T)`	Turn the given possibly-null reference into an Optional, treating non-null as present and null as absent.

Query methods

Each of these are non-static methods on a particular `Optional<T>` value.

Method	Description
`boolean isPresent()`	Returns `true` if this `Optional` contains a non-null instance.
`T get()`	Returns the contained `T` instance, which must be present; otherwise, throws an `IllegalStateException`.
`T or(T)`	Returns the present value in this `Optional`, or if there is none, returns the specified default.
`T orNull()`	Returns the present value in this `Optional`, or if there is none, returns `null`. The inverse operation of `fromNullable`.
`Set<T> asSet()`	Returns an immutable singleton `Set` containing the instance in this `Optional`, if there is one, or otherwise an empty immutable set.

`Optional` provides several more handy utility methods besides these; consult the Javadoc for details.

What's the point?

Besides the increase in readability that comes from giving `null` a *name*, the biggest advantage of Optional is its idiot-proof-ness. It forces you to actively think about the absent case if you want your program to compile at all, since you have to actively unwrap the Optional and address that case. Null makes it disturbingly easy to simply forget things, and though FindBugs helps, we don't think it addresses the issue nearly as well.

This is especially relevant when you're **returning** values that may or may not be "present." You (and others) are far more likely to forget that `other.method(a, b)` could return a null value than you're likely to forget that `a` could be null when you're implementing other.method. Returning `Optional` makes it impossible for callers to forget that case, since they have to unwrap the object themselves for their code to compile.

Convenience methods

Whenever you want a `null` value to be replaced with some default value instead, use `MoreObjects.firstNonNull(T, T)`. As the method name suggests, if both of the inputs are null, it fails fast with a `NullPointerException`. If you are using an `Optional`, there are better alternatives – e.g. `first.or(second)`.

A couple of methods dealing with possibly-null `String` values are provided in `Strings`. Specifically, we provide the aptly named:

- `emptyToNull(String)`
- `isNullOrEmpty(String)`
- `nullToEmpty(String)`

We would like to emphasize that these methods are primarily for interfacing with unpleasant APIs that equate null strings and empty strings. Every time *you* write code that conflates null strings and empty strings, the Guava team weeps. (If null strings and empty strings mean actively different things, that's better, but treating them as the same thing is a disturbingly common code smell.)