Why Do We Need Optional?

What Happens Without Optional?

```
// Consider searching List<Customer> for a Customer with a particular ID.
// Early draft API: Stream.search(Predicate)

Customer customerByID(List<Customer> custList, int custID) {
    return custList.stream()
                   .search(c -> c.getID() == custID);
}

// What if there is no element in the stream matches the predicate?
// Presumably search() returns null.
// customerByID() would then return null.
```

What Happens Without Optional?

```
// Consider searching List<Customer> for a Customer with a particular ID,
// and return the Customer name.

String customerNameByID(List<Customer> custList, int custID) {
    return custList.stream()
                   .search(c -> c.getID() == custID)
                   .getName();
}

// Instead, need to do...
String customerNameByID(List<Customer> custList, int custID) {
    Customer cust = custList.stream()
                            .search(c -> c.getID() == custID);
    return cust != null ? cust.getName() : "UNKNOWN";   // UGH! Cluttered,
}                                                       // easy to forget.
```

Throws NullPointerException if no Customer is found!

Introducing Optional

- Optional<T> added in Java 8
- Can be in one of two states:
  - contains a non-null reference to a T *also called "present"*
  - is empty *also called "absent" (don't say "null")*
- Primitive specializations
  - OptionalInt, OptionalLong, OptionalDouble
- Optional itself is a reference type, and can be null – DON'T

Rule #1: Never, ever, use null for an Optional variable or return value.

Rationale for Optional

Optional is intended to provide a ***limited*** mechanism for library method ***return types*** where there is a clear need to represent “no result,” and where using null for that is ***overwhelmingly likely to cause errors***.

Revisiting Example, Using Optional

```
// Actual Streams API has findFirst() and findAny(), returning non-null Optional<T>.
// Predicate should be passed through a filter() upstream.

String customerNameByID(List<Customer> custList, int custID) {
    return custList.stream()
                   .filter(c -> c.getID() == custID)
                   .findFirst()
                   .getName();
}
```

Error: findFirst() returns an Optional<Customer>, but getName() needs a Customer.

Revisiting Example, Using Optional

```
String customerNameByID(List<Customer> custList, int custID) {
    Optional<Customer> opt = custList.stream()
                                     .filter(c -> c.getID() == custID)
                                     .findFirst();

    return opt ??? getName();
}
```

How do we get the Customer out of the Optional<Customer> to call getName() on it?

Revisiting Example, Using Optional

```
String customerNameByID(List<Customer> custList, int custID) {
    Optional<Customer> opt = custList.stream()
                                     .filter(c -> c.getID() == custID)
                                     .findFirst();

    return opt.get().getName();
}
```

To get the value from an Optional, call get()

But get() throws NoSuchElement-Exception if the Optional is empty. Hardly an improvement!

How To Use Optional

Safely Getting a Value from an Optional

```
// A couple methods on Optional<T>: isPresent() and get()


String customerNameByID(List<Customer> custList, int custID) {
    Optional<Customer> opt = custList.stream()
                                     .filter(c -> c.getID() == custID)
                                     .findFirst();
    return opt.isPresent() ? opt.get().getName() : "UNKNOWN";
}
```

This is safe, but hardly any better than checking for null!

Safely Getting a Value from an Optional

```
// A couple methods on Optional<T>: isPresent() and get()


String customerNameByID(List<Customer> custList, int custID) {
    Optional<Customer> opt = custList.stream()
                                     .filter(c -> c.getID() == custID)
                                     .findFirst();
    return opt.isPresent() ? opt.get().getName() : "UNKNOWN";
}
```

Rule #2: Never use Optional.get() unless you can prove that the Optional is present.

Unfortunately, this just leads people into testing isPresent() before get()...

Safely Getting a Value from an Optional

```
// A couple methods on Optional<T>: isPresent() and get()


String customerNameByID(List<Customer> custList, int custID) {
    Optional<Customer> opt = custList.stream()
                                     .filter(c -> c.getID() == custID)
                                     .findFirst();
    return opt.isPresent() ? opt.get().getName() : "UNKNOWN";
}
```

Rule #2: Never use Optional.get() unless you can prove that the Optional is present.
Rule #3: Prefer alternatives to Optional.isPresent() and Optional.get()

The orElse() Method Family

```
// orElse(default)
Optional<Data> opt = ...
Data data = opt.orElse(DEFAULT_DATA);
```

*Returns the value if present, **or else** a default value*

```
// orElseGet(supplier)
Optional<Data> opt = ...
Data data = opt.orElseGet(Data::new);
```

*Returns the value if present, **or else gets** a default value by calling a supplier*

```
// orElseThrow(exsupplier)
Optional<Data> opt = ...
Data data = opt.orElseThrow(IllegalStateException::new);
```

*Returns the value if present, **or else throws** an exception obtained from a supplier*

The map() Method

```
String customerNameByID(List<Customer> custList, int custID) {
    Optional<Customer> opt = custList.stream()
                                     .filter(c -> c.getID() == custID)
                                     .findFirst();

    return opt.isPresent() ? opt.get().getName() : "UNKNOWN";
}
```

Example: Using map() and orElse()

```
String customerNameByID(List<Customer> custList, int custID) {
    Optional<Customer> opt = custList.stream()
                                     .filter(c -> c.getID() == custID)
                                     .findFirst();

 // return opt.isPresent() ? opt.get().getName() : "UNKNOWN";

    return opt.map(Customer::getName).orElse("UNKNOWN");
}
```

map() – If present, transforms or maps the value into another and returns the result in an Optional; otherwise returns an empty Optional.

Example: Using map() and orElse()

```
String customerNameByID(List<Customer> custList, int custID) {
    Optional<Customer> opt = custList.stream()
                                     .filter(c -> c.getID() == custID)
                                     .findFirst();

 // return opt.isPresent() ? opt.get().getName() : "UNKNOWN";

    return opt.map(Customer::getName).orElse("UNKNOWN");
}
```

orElse() can be chained directly off the result of the map() call to extract the value if present, or the default

Example: Using map() and orElse()

```
String customerNameByID(List<Customer> custList, int custID) {
    return custList.stream()
                   .filter(c -> c.getID() == custID)
                   .findFirst()
                   .map(Customer::getName)
                   .orElse("UNKNOWN");
}
```

The map() and orElse() calls on Optional can be chained directly off the end of a stream pipeline.

The filter() Method

```
// (adapted with some liberties from OpenJDK Layer.java, since superseded)
// Given a Configuration object, establish an invariant
// between its parent Configuration and this Layer's Configuration.

Optional<Configuration> oparent = config.parent();
if (!oparent.isPresent() || oparent.get() != this.config()) {
    throw new IllegalArgumentException()
}
```

The filter() Method

```
// (adapted with some liberties from OpenJDK Layer.java, since superseded)
// Given a Configuration object, establish an invariant
// between its parent Configuration and this Layer's Configuration.

Optional<Configuration> oparent = config.parent();
if (!oparent.isPresent() || oparent.get() != this.config()) {
    throw new IllegalArgumentException()
}

config.parent()
      .filter(c -> c == this.config())
      .orElseThrow(IllegalArgumentException::new);
```

filter() – if absent, returns empty; if present, applies a predicate to the value, returning present if true or empty if false.

The ifPresent() Method

```
// Not to be confused with isPresent()!

// Another example from the JDK:

Optional<Task> oTask = getTask(...);
if (oTask.isPresent()) {
    executor.runTask(oTask.get());
}
```

Note isPresent() and get() calls

The ifPresent() Method

```
// Not to be confused with isPresent()!

// Another example from the JDK:

Optional<Task> oTask = getTask(...);
if (oTask.isPresent()) {
    executor.runTask(oTask.get());
}

// better:
getTask(...).ifPresent(task -> executor.runTask(task));
```

ifPresent() – if present, executes lambda (a Consumer) on the value, otherwise does nothing.

The ifPresent() Method

```
// Not to be confused with isPresent()!

// Another example from the JDK:

Optional<Task> oTask = getTask(...);
if (oTask.isPresent()) {
    executor.runTask(oTask.get());
}

// better:
getTask(...).ifPresent(task -> executor.runTask(task));

// best:
getTask(...).ifPresent(executor::runTask);
```

Method references for the win!!

Additional Methods

- Static factory methods
  - Optional.empty() – returns an empty Optional
  - Optional.of(T) – returns a present Optional containing T
    - T must be non-null
- Optional.equals() and hashCode() – mostly as one would expect
- Technique: unit testing a method that returns Optional

```
assertEquals(Optional.of("expected value"), optionalReturningMethod());
assertEquals(Optional.empty(), optionalReturningMethod());
```

New Optional Methods in Java 9

- Stream<T> Optional.stream()
  - returns a Stream of zero or one value depending on whether the Optional is absent or present
- void Optional.ifPresentOrElse(Consumer<T>, Runnable)
  - calls the consumer on the present value, or calls the runnable if the value is absent
- Optional<T> Optional.or(Supplier<Optional<T>>)
  - if 'this' optional is present, returns it
  - otherwise calls the supplier and returns the Optional it produces

Example: Stream of Optional

```
// Convert List<CustomerID> to List<Customer>, ignoring unknowns

// Java 8
    List<Customer> list = custIDlist.stream()
        .map(Customer::findByID)
        .filter(Optional::isPresent)
        .map(Optional::get)
        .collect(Collectors.toList());

// Java 9 adds Optional.stream(), allowing filter/map to be fused into a flatMap:
    List<Customer> list = custIDlist.stream()
        .map(Customer::findByID)
        .flatMap(Optional::stream)
        .collect(Collectors.toList());
```

Assume findByID() returns Optional<Customer>

Let only present Optionals through

Extract values from them

Optional.stream() allows filter() & map() to be fused into flatMap()

Example: Adapting Between Null and Optional

- Sometimes you need to adapt Optional-using code to code that wants null, or vice-versa
- If you have a nullable reference and you need an Optional

```
Optional<T> opt = Optional.ofNullable(ref)
```

- If you have an Optional and you need a nullable reference

```
opt.orElse(null)
```

  - Otherwise, generally avoid orElse(null)

Use, Abstruse Use, and Abuse

Method Chaining is Cool, But...

```
// BAD

String process(String s) {
    return Optional.ofNullable(s).orElseGet(this::getDefault);
}


// GOOD

String process(String s) {
    return (s != null) ? s : getDefault();
}
```

Rule #4: It's generally a bad idea to create an Optional for the specific purpose of chaining methods from it to get a value.

Avoiding If-Statements is Cool, But...

```
Optional<BigDecimal> first = getFirstValue();
Optional<BigDecimal> second = getSecondValue();

// Add first and second, treating empty as zero, returning an Optional of the sum,
// unless BOTH are empty, in which case return an empty Optional.

Optional<BigDecimal> result = ...
```

http://stackoverflow.com/q/39498338/1441122

Avoiding If-Statements is Cool, But...

```
Optional<BigDecimal> first = getFirstValue();
Optional<BigDecimal> second = getSecondValue();

// Add first and second, treating empty as zero, returning an Optional of the sum,
// unless BOTH are empty, in which case return an empty Optional.

Optional<BigDecimal> result =
    Stream.of(first, second)                          // OK, BUT COULD BE BETTER
          .filter(Optional::isPresent)
          .map(Optional::get)
          .reduce(BigDecimal::add);
```

Clever, and allows any number of Optionals to be combined.

Avoiding If-Statements is Cool, But...

```
Optional<BigDecimal> first = getFirstValue();
Optional<BigDecimal> second = getSecondValue();

// Add first and second, treating empty as zero, returning an Optional of the sum,
// unless BOTH are empty, in which case return an empty Optional.

Optional<BigDecimal> result =
    first.map(b -> second.map(b::add).orElse(b))   // NOT RECOMMENDED
         .map(Optional::of)
         .orElse(second);
```

Even more clever!

Exercise: verify this is correct.

Avoiding If-Statements is Cool, But...

```
Optional<BigDecimal> first = getFirstValue();
Optional<BigDecimal> second = getSecondValue();

// Add first and second, treating empty as zero, returning an Optional of the sum,
// unless BOTH are empty, in which case return an empty Optional.

Optional<BigDecimal> result;

if (!first.isPresent() && !second.isPresent()) {
    result = Optional.empty();
} else {
    result = Optional.of(first.orElse(ZERO).add(second.orElse(ZERO)));
}
```

Not the shortest, or cleverest, but is it the clearest?

Rule #5: If an Optional chain is nested or has an intermediate result of Optional<Optional<T>>, it's probably too complex.

The Problem With Optional.get()

Brian Goetz' biggest Java 8 regret:

> There is a get() method on Optional; we should have never called it get(). We should have called it getOrThrowSomethingHorribleIfTheThingIsEmpty() because everybody calls it thinking, "I am just supposed to call Optional.get()" and they don't realize that that it completely undermines the purpose of using Optional, because it is going to throw [an exception] if the Optional is empty.
>
> On Stack Overflow, every second post that uses Optional misuses Optional.get(), and it's totally my fault, because I should have named it something much more horrible. In your IDE, the get() method pops up, and you say, "oh yeah, that's what I want" and if something with a scarier name popped up, it might make you think, "Which of these get methods do I want? Do I want the one that throws, or do I want the one that returns an alternative?"

JAX 2015 *Fragen und Antworten zu Java 8* with Angelika Langer, at 16:00.
https://jaxenter.de/fragen-und-antworten-zu-java-8-qa-33108

The Problem With Optional.get()

- The get() method is an “attractive nuisance”
  - it’s much less useful than its short name would indicate
  - easy to forget to guard it
  - easy to be misled into poor isPresent() / get() coding style
  - get() is misused in a significant fraction of cases => therefore it’s a bad API
- Plan
  - introduce replacement for get()
  - deprecate get()
    - not for removal
  - deprecation on hold because of warnings it introduces

Rule #2: Never use Optional.get() unless you can prove that the Optional is present.

Rule #3: Prefer alternatives APIs over Optional.isPresent() and Optional.get().

Places Not to Use Optional

- Avoid using Optional in fields
  - fill in replacement value at init time; use "null object" pattern; use actual null
- Avoid using Optional in method parameters
  - it doesn't really work for making parameters optional
  - forces call sites to create Optionals for everything:

```
myMethod(Optional.of("some value"));    // DON'T DO THIS!
myMethod(Optional.empty());
```

- Avoid using Optional in collections
  - usually indicates a design smell of sorts
  - often better ways of representing things

Rule #6: Avoid using Optional in fields, method parameters, and collections.

Places Not to Use Optional

- Remember, Optional is a box!
  - is a separate object, consumes 4x memory of a bare reference
  - potentially adds GC pressure
  - always requires a dependent load, leading to cache misses
  - could turn into a performance/space problem if used frequently
- Don't replace every null with an Optional
  - null can be safe, if it's well controlled
  - null in a private field can be easily checked
  - nullable parameters are ok (if déclassé)
    - library code should take responsibility for checking args

Why Isn't Optional Serializable?

- Background: Value types – Project Valhalla
  - a value is like an object, but it has no notion of identity
  - "codes like a class, works like an int"
  - we eventually want to convert Optional into a value type
- Disclaimer from Optional's javadoc:

  This is a value-based class; use of identity-sensitive operations (including reference equality (==), identity hash code, or synchronization) on instances of Optional may have unpredictable results and should be avoided.

Rule #7: Avoid using identity-sensitive operations on Optionals.

Why Not Use Optional in Fields?

- More a style issue than a correctness issue
  - usually there's a better way to model absence of a value
  - distinguish usage in API vs internal representation
  - use of Optional in fields often arises from slavish desire to eliminate nullable fields
  - remember, eliminating nulls isn't a goal of Optional
- Using Optional in fields...
  - creates another object for every field – 4x memory overhead vs bare reference
  - introduces a dependent load from memory on every field read
  - clutters up your code
  - to what benefit? ability to chain methods?

Why Not Use Optional in Fields?

- Colebourne: *Optional, A Pragmatic Approach*
  - http://blog.joda.org/2015/08/java-se-8-optional-pragmatic-approach.html
  - use nullable fields, getters should return Optional
- Ernst: *Nothing is Better Than the Optional Type*
  - https://homes.cs.washington.edu/~mernst/advice/nothing-is-better-than-optional.html
  - use Nullness Checker
  - glass is one-quarter full

Summary & Conclusion

Optional is intended to provide a ***limited*** mechanism for library method ***return types*** where there is a clear need to represent "no result," and where using null for that is ***overwhelmingly likely to cause errors***.

Summary & Conclusion

- ***Rule #1: Never, ever, use null for an Optional variable or return value.***
- ***Rule #2: Never use Optional.get() unless you can prove that the Optional is present.***
- ***Rule #3: Prefer alternatives to Optional.isPresent() and Optional.get().***
- ***Rule #4: It's generally a bad idea to create an Optional for the specific purpose of chaining methods from it to get a value.***
- ***Rule #5: If an Optional chain has a nested Optional chain, or has an intermediate result of Optional<Optional<T>>, it's probably too complex.***
- ***Rule #6: Avoid using Optional in fields, method parameters, and collections.***
- ***Rule #7: Avoid using identity-sensitive operations on Optionals.***