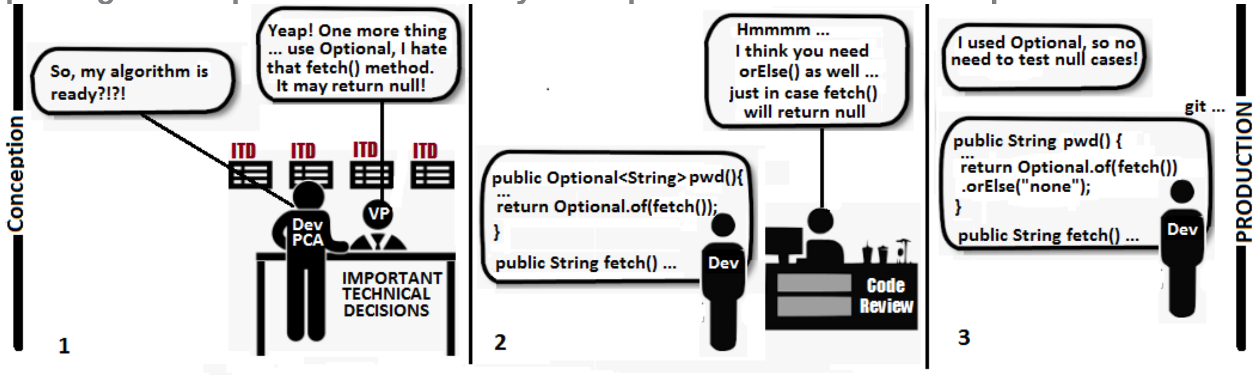
Optional Correctly Is Not Optional

Optionals are not as easy as they seem.

by
Anghel Leonard

.
Nov 28, 18 · Java Zone · Presentation


Conception
So, my algorithm is ready?!?!
Yeap! One more thing ... use Optional, I hate that fetch() method. It may return null!
ITD
ITD
ITD
ITD
Dev PCA
VP
IMPORTANT TECHNICAL DECISIONS
1
Hmmmm ... I think you need orElse() as well ... just in case fetch() will return null
Dev
Code Review
2
I used Optional, so no need to test null cases!
git ...
Dev
3
PRODUCTION

```
public Optional<String> pwd(){
...
return Optional.of(fetch());
}
public String fetch() ...
```

```
public String pwd() {
...
return Optional.of(fetch())
.orElse("none");
}
public String fetch() ...
```

So, what did the VP of TPM do wrong? Additionally, what did the code reviewer and developer do wrong? Well, the answers are at the end of this article, but if you didn't intuit that the code that will go into production is still open for NPE, then maybe this article is for you. And, yes, somehow the code passed the testing phase as well.

The best way to use things is to exploit them for what they have been created and tested for in the first place. Java 8 `Optional` is not an exception to this rule. The purpose of Java 8 `Optional` is clearly defined by Brian Goetz, Java's language architect:

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

So, how to use `Optional` the way it was intended? Typically, we learn to use things by learning how to not use them, and, somehow, this is the approach here as well. So, let's tackle this topic via 26 items. This is a suite of items that try to address `Optional` in your code through an elegant and painless approach.

Item 1: Never Assign Null to an Optional Variable

Avoid:

```
// AVOID
public Optional<Cart> fetchCart() {
    Optional<Cart> emptyCart = null;
    ...
}
```

Prefer:

```
// PREFER
public Optional<Cart> fetchCart() {
    Optional<Cart> emptyCart = Optional.empty();
    ...
}
```

Prefer `Optional.empty()` to initialize an `Optional` instead of a `null` value. `Optional` is just a container/box and it is pointless to initialize it with `null`.

Item 2: Ensure That an Optional Has a Value Before Calling Optional.get()

If, for any reason, you decide that `Optional.get()` will make your day, then don't forget that you must prove that the `Optional` value is present before this call. Typically, you will do it by adding a check (condition) based on the`Optional.isPresent()`method. The `isPresent()-get()`pair has a bad reputation (check further items for alternatives), but if this is your chosen path, then don't forget about the`isPresent()`part. Nevertheless, keep in mind that`Optional.get()`is prone to be deprecated at some point.

Avoid:

```
// AVOID
Optional<Cart> cart = ... ; // this is prone to be empty
...
// if "cart"is empty then this code will throw a java.util.NoSuchElementExcep
tion
Cart myCart = cart.get();
```

Prefer:

```
// PREFER
if (cart.isPresent()) {
    Cart myCart = cart.get();
    ... // do something with "myCart"
} else {
    ... // do something that doesn't call cart.get()
}
```

Item 3: When No Value Is Present, Set/Return an Already-Constructed Default Object Via the Optional.orElse() Method

Using the`Optional.orElse()`method represents an elegant alternative to the`isPresent()-get()`pair for setting/returning a value. The important thing here is that the parameter of`orElse()`is evaluated even when having a non-empty`Optional`. This means that it will be evaluated even if it will not be used, which is a performance penalty. In this context, use`orElse()`only when the parameter (the default object) is already constructed. In other situations, rely on item 4.

Avoid:

```
// AVOID
public static final String USER_STATUS = "UNKNOWN";
...
public String findUserStatus(long id) {
    Optional<String> status = ... ; // prone to return an empty Optional
    if (status.isPresent()) {
        return status.get();
    } else {
        return USER_STATUS;
    }
}
```

Prefer:

```
// PREFER
public static final String USER_STATUS = "UNKNOWN";
...
public String findUserStatus(long id) {
    Optional<String> status = ... ; // prone to return an empty Optional
    return status.orElse(USER_STATUS);
}
```

Item 4: When No Value Is Present, Set/Return a Non-Existent Default Object Via the Optional.orElseGet() Method

Using the`Optional.orElseGet()`method represents another elegant alternative to the`isPresent()-get()`pair for setting/returning a value. The important thing here is that the parameter of`orElseGet()`is a Java 8,`Supplier`. This means that the`Supplier`method passed as an argument is only executed when an`Optional`value is not present. So, this is useful to avoid the`orElse()`performance penalty of creating objects and executing code that we don't need when an`Optional`value is present.

Avoid:

```
// AVOID
public String computeStatus() {
    ... // some code used to compute status
}
public String findUserStatus(long id) {
    Optional<String> status = ... ; // prone to return an empty Optional
    if (status.isPresent()) {
        return status.get();
    } else {
        return computeStatus();
    }
}
```

Also, avoid:

```
// AVOID
public String computeStatus() {
    ... // some code used to compute status
}
public String findUserStatus(long id) {
    Optional<String> status = ... ; // prone to return an empty Optional
    // computeStatus() is called even if "status" is not empty
    return status.orElse(computeStatus());
}
```

Prefer:

```
// PREFER
public String computeStatus() {
    ... // some code used to compute status
}
public String findUserStatus(long id) {
    Optional<String> status = ... ; // prone to return an empty Optional
    // computeStatus() is called only if "status" is empty
    return status.orElseGet(this::computeStatus);
}
```

Item 5: When No Value Is Present, Throw a java.util.NoSuchElementException Exception Via orElseThrow() Since Java 10

Using the `Optional.orElseThrow()` method represents another elegant alternative to the `isPresent()`-`get()` pair. Sometimes, when an `Optional` value is not present, all you want to do is to throw a `java.util.NoSuchElementException` exception. Starting with Java 10, this can be done via the `orElseThrow()` method without arguments. For Java 8 and 9, please consider item 6.

Avoid:

```
// AVOID
public String findUserStatus(long id) {
    Optional<String> status = ... ; // prone to return an empty Optional
    if (status.isPresent()) {
        return status.get();
    } else {
        throw new NoSuchElementException();
    }
}
```

Prefer:

```
// PREFER
public String findUserStatus(long id) {
    Optional<String> status = ... ; // prone to return an empty Optional
    return status.orElseThrow();
}
```

Item 6: When No Value Is Present, Throw an Explicit Exception Via orElseThrow(Supplier<? extends X> exceptionSupplier)

In Java 10, for `java.util.NoSuchElementException`, use `orElseThrow()`, as shown in item 5.

Using the`Optional.orElseThrow(Supplier<? extends X> exceptionSupplier)`method represents another elegant alternative to the`isPresent()-get()`pair. Sometimes, when an`Optional`value is not present, all you want to do is to throw an explicit exception. Starting with Java 10, if that exception is`java.util.NoSuchElementException`then simply rely on`orElseThrow()`method without arguments - item 5.

Avoid:

```
// AVOID
public String findUserStatus(long id) {
    Optional<String> status = ... ; // prone to return an empty Optional
    if (status.isPresent()) {
        return status.get();
    } else {
        throw new IllegalStateException();
    }
}
```

Prefer:

```
// PREFER
public String findUserStatus(long id) {
    Optional<String> status = ... ; // prone to return an empty Optional
    return status.orElseThrow(IllegalStateException::new);
}
```

Item 7: When You Have an Optional and Need a Null Reference, Use orElse(null)

When you have an`Optional`and need a`null`reference, then use `orElse(null)`. Otherwise, avoid `orElse(null)`.

A typical scenario for using`orElse(null)`occurs when we have an`Optional`and we need to call a method that accepts`null`references in certain cases. For example, let's look at `Method.invoke()` from the Java Reflection API. The first argument of this method is the object instance on which this particular method is to be invoked. If the method is`static`, the first argument should be`null`.

Avoid:

```
// AVOID
Method myMethod = ... ;
...
// contains an instance of MyClass or empty if "myMethod" is static
Optional<MyClass> instanceMyClass = ... ;
...
if (instanceMyClass.isPresent()) {
    myMethod.invoke(instanceMyClass.get(), ...);
} else {
    myMethod.invoke(null, ...);
}
```

Prefer:

```
// PREFER
Method myMethod = ... ;
...
// contains an instance of MyClass or empty if "myMethod" is static
Optional<MyClass> instanceMyClass = ... ;
...
myMethod.invoke(instanceMyClass.orElse(null), ...);
```

Item 8: Consume an Optional if it Is Present. Do Nothing if it Is Not Present. This Is a Job For Optional.ifPresent().

The`Optional.ifPresent()`is a good alternative for`isPresent()-get()`pair when you just need to consume the value. If no value is present then do nothing.

Avoid:

```
// AVOID
Optional<String> status = ... ;
...
if (status.isPresent()) {
    System.out.println("Status: " + status.get());
}
```

Prefer:

```
// PREFER
Optional<String> status ... ;
```

```
...
status.ifPresent(System.out::println);
```

Item 9: Consume an Optional if it Is Present. If it Is Not Present, Then Execute an Empty-Based Action. This Is a Job For Optional.ifPresentElse(), Java 9.

Starting with Java 9, the`Optional.ifPresentOrElse()`is a good alternative for`isPresent()-get()`pair. This is similar with the `ifPresent()`method only that it covers the`else`branch as well.

Avoid:

```
// AVOID
Optional<String> status = ... ;
if(status.isPresent()) {
    System.out.println("Status: " + status.get());
} else {
    System.out.println("Status not found");
}
```

Prefer:

```
// PREFER
Optional<String> status = ... ;
status.ifPresentOrElse(
    System.out::println,
    () -> System.out.println("Status not found")
);
```

Item 10: When the Value Is Present, Set/Return That Optional. When No Value Is Present, Set/Return the Other Optional. This Is a Job For Optional.or(), Java 9.

Sometimes, for non-empty `Optional`, we want to set/return that `Optional`. And when our`Optional`is empty, we want to execute some other action that also returns an `Optional`. The`orElse()`and`orElseGet()`methods cannot accomplish this since both will return the **unwrapped** values. It is time to introduce the Java 9,`Optional.or()`method, which is capable of returning an`Optional`describing the value. Otherwise, it returns an`Optional`produced by the supplying function.

Avoid:

```
// AVOID
public Optional<String> fetchStatus() {
    Optional<String> status = ... ;
    Optional<String> defaultStatus = Optional.of("PENDING");
    if (status.isPresent()) {
        return status;
    } else {
        return defaultStatus;
    }
}
```

Also, avoid:

```
// AVOID
public Optional<String> fetchStatus() {
    Optional<String> status = ... ;
    return status.orElseGet(() -> Optional.<String>of("PENDING"));
}
```

Prefer:

```
// PREFER
public Optional<String> fetchStatus() {
    Optional<String> status = ... ;
    Optional<String> defaultStatus = Optional.of("PENDING");
    return status.or(() -> defaultStatus);
    // or, without defining "defaultStatus"
    return status.or(() -> Optional.of("PENDING"));
}
```

Item 11: Optional.orElse/ orElseXXX Are a Perfect Replacement for isPresent()-get() Pair in Lambdas

This can be used to obtain chained lambdas instead of disrupted code. Consider `ifPresent()` and `ifPresentOrElse()` in Java 9 or the `or()` method in Java 9, as well.

Some operations specific to lambdas are returning an`Optional`(e.g.,`findFirst()`, `findAny()`, `reduce()`,...). Trying to use this`Optional`via the`isPresent()-get()`pair is a clumsy approach since it may require you to break the chain of lambdas, pollute the code with`if`statements, and consider continuing the chain. In such cases, the`orElse()`and`orElseXXX()`are very handy since they can be chained directly in the chain of lambda expressions and avoid disrupted code.

Example 1

Avoid:

```
// AVOID
List<Product> products = ... ;
Optional<Product> product = products.stream()
    .filter(p -> p.getPrice() < price)
    .findFirst();
if (product.isPresent()) {
    return product.get().getName();
} else {
    return "NOT FOUND";
}
```

Also, avoid:

```
// AVOID
List<Product> products = ... ;
Optional<Product> product = products.stream()
    .filter(p -> p.getPrice() < price)
    .findFirst();
return product.map(Product::getName)
    .orElse("NOT FOUND");
```

Prefer:

```
// PREFER
List<Product> products = ... ;
return products.stream()
    .filter(p -> p.getPrice() < price)
    .findFirst()
    .map(Product::getName)
    .orElse("NOT FOUND");
```

Example 2

Avoid:

```
// AVOID
Optional<Cart> cart = ... ;
Product product = ... ;
...
if(!cart.isPresent() ||
   !cart.get().getItems().contains(product)) {
    throw new NoSuchElementException();
}
```

Prefer:

```
// PREFER
Optional<Cart> cart = ... ;
```

```
Product product = ... ;
...
cart.filter(c -> c.getItems().contains(product)).orElseThrow();
```

Item 12: Avoid Chaining Optional's Methods With the Single Purpose of Getting a Value

Sometimes, we tend to "over-use" things. Meaning that we have a thing, like `Optional`, and we see a use case for it everywhere. In the case of `Optional`, a common scenario involves chaining its methods for the single purpose of getting a value. Avoid this practice and rely on simple and straightforward code.

Avoid:

```
// AVOID
public String fetchStatus() {
    String status = ... ;
    return Optional.ofNullable(status).orElse("PENDING");
}
```

Prefer:

```
// PREFER
public String fetchStatus() {
    String status = ... ;
    return status == null ? "PENDING" : status;
}
```

Item 13: Do Not Declare Any Field of Type Optional

Do not use `Optional` in methods (including `setters`) or constructors arguments.

Remember that`Optional` was not intended to be used for fields and it doesn't implement`Serializable`. The`Optional`class is definitively not intended for use as a property of a Java Bean.

Avoid:

```
// AVOID
public class Customer {
    [access_modifier] [static] [final] Optional<String> zip;
    [access_modifier] [static] [final] Optional<String> zip = Optional.empty(
);
```

```
    ...
}
```

Prefer:

```
// PREFER
public class Customer {
    [access_modifier] [static] [final] String zip;
    [access_modifier] [static] [final] String zip = "";
    ...
}
```

Item 14: Do Not Use Optional in Constructors Arguments

Do not use `Optional` as fields or in methods' (including `setters`) arguments.

This is another usage against`Optional`intention. `Optional`wraps objects with another level of abstraction, which, in that case, simply adds extra boilerplate code.

Avoid:

```
// AVOID
public class Customer {
    private final String name;               // cannot be null
    private final Optional<String> postcode; // optional field, thus may be n
ull
    public Customer(String name, Optional<String> postcode) {
        this.name = Objects.requireNonNull(name, () -> "Name cannot be null")
;
        this.postcode = postcode;
    }
    public Optional<String> getPostcode() {
        return postcode;
    }
    ...
}
```

Prefer:

```
// PREFER
public class Customer {
    private final String name;     // cannot be null
    private final String postcode; // optional field, thus may be null
    public Cart(String name, String postcode) {
        this.name = Objects.requireNonNull(name, () -> "Name cannot be null")
;
        this.postcode = postcode;
    }
    public Optional<String> getPostcode() {
        return Optional.ofNullable(postcode);
    }
    ...
}
```

As you can see, now the getter returns an `Optional`. Don't take this example as a rule for transforming all your getters like this. Most of the time, getters return collections or arrays, and in that case, they prefer returning empty collections/arrays instead of `Optional`. Use this technique and keep in mind this Brian Goetz statement:

I think routinely using it as a return value for getters would definitely be over-use.

Item 15: Do Not Use Optional in Setters Arguments

`Optional` is not intended for use as a property of a Java Bean or as the persistent property type. `Optional` is not`Serializable`.

Using`Optional`in setters is another anti-pattern. Commonly, I saw it used as the persistent property type (map an entity attribute as `Optional`). However, using`Optional`in *Domain Model* entities is possible, as shown in the following example.

Avoid:

```
// AVOID
@Entity
public class Customer implements Serializable {
    private static final long serialVersionUID = 1L;
    ...
    @Column(name="customer_zip")
    private Optional<String> postcode; // optional field, thus may be null
     public Optional<String> getPostcode() {
       return postcode;
     }
     public void setPostcode(Optional<String> postcode) {
       this.postcode = postcode;
     }
     ...
}
```

Prefer:

```
// PREFER
@Entity
public class Customer implements Serializable {
    private static final long serialVersionUID = 1L;
    ...
    @Column(name="customer_zip")
    private String postcode; // optional field, thus may be null
    public Optional<String> getPostcode() {
      return Optional.ofNullable(postcode);
    }
    public void setPostcode(String postcode) {
       this.postcode = postcode;
    }
    ...
}
```

Item 16: Do Not Use Optional in Methods Arguments

Don't force call sites to create `Optionals`. Do not use `Optional` as fields or in setters and constructors' arguments.

Using`Optional`in methods arguments is another common mistake. This would lead to code that is unnecessarily complicated. Take responsibility for checking arguments instead of force call sites to create `Optional`s. This practice clutters the code and may cause dependence. In time, you will use it everywhere. Keep in mind that`Optional`is just another object (a container) and is not cheap — it consumes 4x the memory of a bare reference! But you may also like to check this out and decide, depending on the case.

Avoid:

```
// AVOID
public void renderCustomer(Cart cart, Optional<Renderer> renderer,
                           Optional<String> name) {
    if (cart == null) {
        throw new IllegalArgumentException("Cart cannot be null");
    }
    Renderer customerRenderer = renderer.orElseThrow(
        () -> new IllegalArgumentException("Renderer cannot be null")
    );
    String customerName = name.orElseGet(() -> "anonymous");
    ...
}
// call the method - don't do this
renderCustomer(cart, Optional.<Renderer>of(CoolRenderer::new), Optional.empty
());
```

Prefer:

```
// PREFER
public void renderCustomer(Cart cart, Renderer renderer, String name) {
    if (cart == null) {
        throw new IllegalArgumentException("Cart cannot be null");
    }
```

```java
    if (renderer == null) {
        throw new IllegalArgumentException("Renderer cannot be null");
    }
    String customerName = Objects.requireNonNullElseGet(name, () -> "anonymou
s");
    ...
}
// call this method
renderCustomer(cart, new CoolRenderer(), null);
```

Also, prefer (rely on`NullPointerException`):

```java
// PREFER
public void renderCustomer(Cart cart, Renderer renderer, String name) {
    Objects.requireNonNull(cart, "Cart cannot be null");
    Objects.requireNonNull(renderer, "Renderer cannot be null");
    String customerName = Objects.requireNonNullElseGet(name, () -> "anonymou
s");
    ...
}
// call this method
renderCustomer(cart, new CoolRenderer(), null);
```

Also, prefer (avoid`NullPointerException`and use`IllegalArgumentException`or other exception):

```java
// PREFER
// write your own helper
public final class MyObjects {
    private MyObjects() {
        throw new AssertionError("Cannot create instances for you!");
    }
    public static <T, X extends Throwable> T requireNotNullOrElseThrow(T obj,
        Supplier<? extends X> exceptionSupplier) throws X {
        if (obj != null) {
            return obj;
        } else {
            throw exceptionSupplier.get();
        }
    }
}
public void renderCustomer(Cart cart, Renderer renderer, String name) {
    MyObjects.requireNotNullOrElseThrow(cart,
                () -> new IllegalArgumentException("Cart cannot be null"));
    MyObjects.requireNotNullOrElseThrow(renderer,
                () -> new IllegalArgumentException("Renderer cannot be null")
);
    String customerName = Objects.requireNonNullElseGet(name, () -> "anonymou
s");
    ...
}
// call this method
renderCustomer(cart, new CoolRenderer(), null);
```

Item 17: Do Not Use Optional to Return Empty Collections or Arrays

Favor returning an empty collection/array. With this in mind, rely on `Collections.emptyList()`, `emptyMap()`, and `emptySet()`.

In order to keep the code clean and lightweight avoid returning `Optional` for `null` or empty collections/arrays. Prefer returning an empty array or collection.

Avoid:

```java
// AVOID
public Optional<List<String>> fetchCartItems(long id) {
    Cart cart = ... ;
    List<String> items = cart.getItems(); // this may return null
    return Optional.ofNullable(items);
}
```

Prefer:

```java
// PREFER
public List<String> fetchCartItems(long id) {
    Cart cart = ... ;
    List<String> items = cart.getItems(); // this may return null
    return items == null ? Collections.emptyList() : items;
}
```

Item 18: Avoid Using Optional in Collections

Usually, there are better ways to represent things. This approach can be a design smell.

Using `Optional` in collections can be a design smell. Take another 30 minutes to think of the problem and I'm sure you will find better ways. But, probably, the most used argument to sustain this approach refers to `Map`. It sounds like this: *so, a* `Map` *returns* `null` *if there is no mapping for a key or if* `null` *is mapped to the key, so I cannot distinguish it if the key is not present or is a missing value. I will wrap the values via* `Optional.ofNullable` *and done*! Well, what you will do if your `Map` of `Optional` *Goodies* will be populated with `null` values, absent `Optional` values, or even `Optional` objects that contains something else, but not your *Goodies*? Haven't you just nested the initial problem into one more layer? How about the performance penalty? `Optional`is not cost-free, it is just another object that consumes memory and needs to be collected.

Avoid:

```
// AVOID
Map<String, Optional<String>> items = new HashMap<>();
items.put("I1", Optional.ofNullable(...));
items.put("I2", Optional.ofNullable(...));
...
Optional<String> item = items.get("I1");
if (item == null) {
    System.out.println("This key cannot be found");
} else {
    String unwrappedItem = item.orElse("NOT FOUND");
    System.out.println("Key found, Item: " + unwrappedItem);
}
```

Prefer (Java 8):

```
//PREFER
Map<String, String> items = new HashMap<>();
items.put("I1", "Shoes");
items.put("I2", null);
...
// get an item
String item = get(items, "I1");  // Shoes
String item = get(items, "I2");  // null
String item = get(items, "I3");  // NOT FOUND
private static String get(Map<String, String> map, String key) {
  return map.getOrDefault(key, "NOT FOUND");
}
```

You can also rely on other approaches, such as:

- `containsKey()`method
- Trivial implementation by extending`HashMap`
- Java 8`computeIfAbsent()`method
- Apache Commons`DefaultedMap`

By extrapolating this example to other collections, we can conclude that there are always better solutions than using`Optional`in collections.

And, this is even worse:

```
Map<Optional<String>, String> items = new HashMap<>();
Map<Optional<String>, Optional<String>> items = new HashMap<>();
```

Item 19: Do Not Confuse Optional.of() and Optional.ofNullable()

Confusing or mistakenly using `Optional.of`instead of`Optional.ofNullable`, or vice-versa, can lead to unpleasant issues. As a key here, keep in mind that`Optional.of(null)`will throw`NullPointerException`, while`Optional.ofNullable(null)`will result in an `Optional.empty`.

Use`Optional.of`instead of`Optional.ofNullable`example

Avoid:

```
// AVOID
public Optional<String> fetchItemName(long id) {
    String itemName = ... ; // this may result in null
    ...
    return Optional.of(itemName); // this throws NPE if "itemName" is null :(
}
```

Prefer:

```
// PREFER
public Optional<String> fetchItemName(long id) {
    String itemName = ... ; // this may result in null
    ...
    return Optional.ofNullable(itemName); // no risk for NPE
}
```

Use`Optional.ofNullable`instead of`Optional.of`example

Avoid:

```
// AVOID
return Optional.ofNullable("PENDING"); // ofNullable doesn't add any value
```

Prefer:

```
// PREFER
return Optional.of("PENDING"); // no risk to NPE
```

Item 20: Avoid Optional <T> and Choose Non-Generic OptionalInt, OptionalLong, or OptionalDouble

Unless you have a specific need for boxed primitives, avoid `Optional< T >` and select the non-generic `OptionalInt`, `OptionalLong`, or `OptionalDouble`.

Boxing and *unboxing* are expensive operations that are prone to induce performance penalties. In order to eliminate this risk, we can rely on `OptionalInt`,`OptionalLong` , and `OptionalDouble`. These are wrappers for primitive types`int`,`long`, and`double`.

Avoid (use it only if you need boxed primitives):

```
// AVOID
Optional<Integer> price = Optional.of(50);
Optional<Long> price = Optional.of(50L);
Optional<Double> price = Optional.of(50.43d);
```

Prefer:

```
// PREFER
OptionalInt price = OptionalInt.of(50);           // unwrap via getAsInt()
OptionalLong price = OptionalLong.of(50L);        // unwrap via getAsLong()
OptionalDouble price = OptionalDouble.of(50.43d); // unwrap via getAsDouble()
```

Item 21: There Is No Need to Unwrap Optionals for Asserting Equality

Having two`Optionals`in an`assertEquals()`doesn't require unwrapped values. This is applicable because`Optional#equals()`compares the wrapped values, not the`Optional`objects.

`Optional.equals()` source code:

```
@Override
public boolean equals(Object obj) {
    if (this == obj) {
        return true;
    }
    if (!(obj instanceof Optional)) {
        return false;
    }
    Optional<?> other = (Optional<?>) obj;
    return Objects.equals(value, other.value);
}
```

Avoid:

```
// AVOID
Optional<String> actualItem = Optional.of("Shoes");
Optional<String> expectedItem = Optional.of("Shoes");
assertEquals(expectedItem.get(), actualItem.get());
```

Prefer:

```
// PREFER
```

```
Optional<String> actualItem = Optional.of("Shoes");
Optional<String> expectedItem = Optional.of("Shoes");
assertEquals(expectedItem, actualItem);
```

Item 22: Transform Values Via Map() and flatMap()

The `Optional.map()` and `Optional.flatMap()` are very convenient approaches for transforming the `Optional` value. The `map()` method applies the *function* argument to the value, then returns the result wrapped in an `Optional`, while, by comparison, the `flatMap()` method takes a *function* argument that is applied to an `Optional` value, and then returns the result directly.

Using `map()`

Example 1

Avoid:

```
// AVOID
Optional<String> lowername ...; // may be empty
// transform name to upper case
Optional<String> uppername;
if (lowername.isPresent()) {
    uppername = Optional.of(lowername.get().toUpperCase());
} else {
    uppername = Optional.empty();
}
```

Prefer:

```
// PREFER
Optional<String> lowername ...; // may be empty
// transform name to upper case
Optional<String> uppername = lowername.map(String::toUpperCase);
```

Example 2

Avoid:

```
// AVOID
List<Product> products = ... ;
Optional<Product> product = products.stream()
    .filter(p -> p.getPrice() < 50)
    .findFirst();
String name;
if (product.isPresent()) {
    name = product.get().getName().toUpperCase();
} else {
    name = "NOT FOUND";
}
// getName() return a non-null String
public String getName() {
    return name;
}
```

Prefer:

```
// PREFER
List<Product> products = ... ;
String name = products.stream()
    .filter(p -> p.getPrice() < 50)
    .findFirst()
    .map(Product::getName)
    .map(String::toUpperCase)
    .orElse("NOT FOUND");
// getName() return a String
public String getName() {
    return name;
}
```

Using `flatMap()`

Avoid:

```
// AVOID
List<Product> products = ... ;
Optional<Product> product = products.stream()
    .filter(p -> p.getPrice() < 50)
    .findFirst();
String name = null;
if (product.isPresent()) {
    name = product.get().getName().orElse("NOT FOUND").toUpperCase();
}
// getName() return an Optional
public Optional<String> getName() {
    return Optional.ofNullable(name);
}
```

Prefer:

```
// PREFER
List<Product> products = ... ;
String name = products.stream()
    .filter(p -> p.getPrice() < 50)
    .findFirst()
    .flatMap(Product::getName)
    .map(String::toUpperCase)
    .orElse("NOT FOUND");
// getName() return an Optional
public Optional<String> getName() {
    return Optional.ofNullable(name);
}
```

Item 23: Reject Wrapped Values Based on a Predefined Rule Using filter()

Pass a predicate (the condition) as an argument and get an `Optional` object. Use the initial `Optional` if the condition is met, and use an empty `Optional` if the condition is not met.

Using`filter()`to accept or reject a wrapped value is a very convenient approach since it can be accomplished without explicitly unwrapping the value.

Avoid:

```
// AVOID
public boolean validatePasswordLength(User userId) {
    Optional<String> password = ...; // User password
    if (password.isPresent()) {
        return password.get().length() > 5;
    }
    return false;
}
```

Prefer:

```
// PREFER
public boolean validatePasswordLength(User userId) {
    Optional<String> password = ...; // User password
    return password.filter((p) -> p.length() > 5).isPresent();
}
```

Item 24: Do We Need to Chain the Optional API With the Stream API?

If so, we then use the `Optional.stream()` method.

Starting with Java 9, we can treat the`Optional`instance as a`Stream`by applying`Optional.stream()`method. This is useful when you need to chain the Optional API with the Stream API. This method creates a`Stream`of one element or an empty`Stream`(if`Optional`is not present). Further, we can use all the methods that are available in the Stream API.

Avoid:

```
// AVOID
public List<Product> getProductList(List<String> productId) {
    return productId.stream()
```

```
        .map(this::fetchProductById)
        .filter(Optional::isPresent)
        .map(Optional::get)
        .collect(toList());
}
public Optional<Product> fetchProductById(String id) {
    return Optional.ofNullable(...);
}
```

Prefer:

```
// PREFER
public List<Product> getProductList(List<String> productId) {
    return productId.stream()
        .map(this::fetchProductById)
        .flatMap(Optional::stream)
        .collect(toList());
}
public Optional<Product> fetchProductById(String id) {
    return Optional.ofNullable(...);
}
```

Practically,`Optional.stream()` allows us to replace`filter()` and `map()` with `flatMap()`.

Also, we can convert`Optional`to `List`:

```
public static <T> List<T> convertOptionalToList(Optional<T> optional) {
    return optional.stream().collect(toList());
}
```

Item 25: Avoid Using Identity-Sensitive Operations on Optionals

This includes reference equality (==), identity hash-based, or synchronization.

The`Optional`class is a value-based class as `LocalDateTime`.

Avoid:

```
// AVOID
Product product = new Product();
Optional<Product> op1 = Optional.of(product);
Optional<Product> op2 = Optional.of(product);
// op1 == op2 => false, expected true
if (op1 == op2) { ...
```

Prefer:

```
// PREFER
Product product = new Product();
```

```
Optional<Product> op1 = Optional.of(product);
Optional<Product> op2 = Optional.of(product);
// op1.equals(op2) => true,expected true
if (op1.equals(op2)) { ...
```

Never do:

```
// NEVER DO
Optional<Product> product = Optional.of(new Product());
synchronized(product) {
    ...
}
```

Item 26: Return a boolean If The Optional Is Empty. Prefer Java 11, Optional.isEmpty()

Starting with Java 11, we can easily return a`true`if an`Optional`is empty via the`isEmpty()`method.

Avoid (Java 11+):

```
// AVOID (Java 11+)
public Optional<String> fetchCartItems(long id) {
    Cart cart = ... ; // this may be null
    ...
    return Optional.ofNullable(cart);
}
public boolean cartIsEmpty(long id) {
    Optional<String> cart = fetchCartItems(id);
    return !cart.isPresent();
}
```

Prefer (Java 11+):

```
// PREFER (Java 11+)
public Optional<String> fetchCartItems(long id) {
    Cart cart = ... ; // this may be null
    ...
    return Optional.ofNullable(cart);
}
public boolean cartIsEmpty(long id) {
    Optional<String> cart = fetchCartItems(id);
    return cart.isEmpty();
}
```

Well, that's it! Looks like using`Optional`correctly is not as easy as it might seem at first glance. Mainly,`Optional`was intended to be used as a return type and for combining it with streams (or methods that return `Optional`) to build fluent APIs. But, there are several corner-cases and temptations that can be considered traps that will downgrade the quality of your code or even causes unexpected behaviors. Having these 26 items under your toolbelt will be very useful to avoid these traps, but keep in mind what Oliver Wendell Holmes, Sr. once said:

The young man knows the rules, but the old man knows the exceptions.

Do you remember the little cartoon at the beginning of this article? What the VP of TPM did wrong? Well, his advice of using`Optional`is correct only that there is a lack of explanation. Is not clear if he is asking the developer to refactor the`fetch()`method in order to return `Optional`, or to use it as it is, but to wrap its call in an `Optional`. Obviously, the developer understood the second. What did the developer and code reviewer do wrong? Both are missing the fact that `Optional.of(fetch())`will throw NPE if`fetch()`returns `null`. Moreover, the final code is chaining `Optional`methods`of()`and`orElse()`with the single purpose of returning a value instead of returning based on a simple`null`check (Item 12). In the end, the code is still prone to NPE.

One of the acceptable approaches consists in refactoring the`fetch()`method to return`Optional`and from the `pwd()`method to return, `fetch().orElse("none")`.

Hope this helps! Let us know your thoughts in the comments section below.