:Optional API:

[java.util.Optional<T>](http://download.java.net/jdk8/docs/api/java/util/Optional.html) in Java 8 is a poor cousin of [scala.Option[T]](http://www.scala-lang.org/api/current/index.html" \l "scala.Option) and [Data.Maybe in Haskell](http://hackage.haskell.org/packages/archive/base/4.2.0.1/doc/html/Data-Maybe.html). But this doesn’t mean it’s not useful. If this concept is new to you, imagine Optional as a container that may or may not contain some value. Just like all references in Java can point to some object or be null, Option may enclose some (non-null!) reference or be empty.  
  
Turns out that the analogy between Optional and nullable references is quite sensible. Optional was introduced in Java 8 so obviously it is not used throughout the standard Java library - and never will be for the backward compatibility reasons. But I recommend you at least giving it a try and using it whenever you have nullable references. Optional instead of plain null is statically checked at compile time and much more informative as it clearly indicates that a given variable may be present or not. Of course it requires some discipline - you should never assign null to any variable any more.  
  
Usage of *option* (*maybe*) pattern is quite controversial and I am not going to step into this discussion. Instead I present you with few use-cases of null and how they can be retrofitted to Optional<T>. In the following examples given variables and types are used:

public void print(String s) {

System.out.println(s);

}

String x = //...

Optional<String> opt = //...

x is a String that *may* be null, opt is never null, but may or may not contain some value (*present* or *empty*). There are few ways of creating Optional:

opt = Optional.of(notNull);

opt = Optional.ofNullable(mayBeNull);

opt = Optional.empty();

In the first case Optional *must* contain not null value and will throw an exception if null is passed. ofNullable() will either return empty or present (set) Optional. empty() always return empty Optional, corresponding to null. It’s a singleton because Optional<T> is immutable.

**ifPresent() - do something when Optional is set**

Tedious if statement:

if (x != null) {

print(x);

}

can be replaced with higher-order function ifPresent():

opt.ifPresent(x -> print(x));

opt.ifPresent(this::print);

The latter syntax (method reference) can be used when lambda argument (String x) matches function formal parameters.

**filter() - reject (filter out) certain Optional values.**

Sometimes you want to perform some action not only when a reference is set but also when it meets certain condition:

if (x != null && x.contains("ab")) {

print(x);

}

This can be replaced with Optional.filter() that turns present (set) Optional to empty Optional if underlying value does not meet given predicate. If input Optional was empty, it is returned as-is:

opt.

filter(x -> x.contains("ab")).

ifPresent(this::print);

This is equivalent to more imperative:

if(opt.isPresent() && opt.get().contains("ab")) {

print(opt.get());

}

**map() - transform value if present**

Very often you need to apply some transformation on a value, but only if it’s not null (avoiding NullPointerException):

if (x != null) {

String t = x.trim();

if (t.length() > 1) {

print(t);

}

}

This can be done in much more declarative way using map():

opt.

map(String::trim).

filter(t -> t.length() > 1).

ifPresent(this::print);

This becomes tricky. Optional.map() applies given function on a value inside Optional - but only if Optional is present. Otherwise nothing happens and empty() is returned. Remember that the transformation is type-safe - look at generics here:

Optional<String> opt = //...

Optional<Integer> len = opt.map(String::length);

If Optional<String> is present Optional<Integer> len is present as well, wrapping length of a String. But if opt was empty, map() over it does nothing except changing generic type.

**orElse()/orElseGet() - turning empty Optional<T>to default T**

At some point you may wish to unwrap Optional and get a hold of real value inside. But you can’t do this if Optional is empty. Here is a pre-Java 8 way of handling such scenario:

int len = (x != null)? x.length() : -1;

With Optional we can say:

int len = opt.map(String::length).orElse(-1);

There is also a version that accepts [Supplier<T>](http://download.java.net/jdk8/docs/api/java/util/function/Supplier.html) if computing default value is slow, expensive or has side-effects:

int len = opt.

map(String::length).

orElseGet(() -> slowDefault()); //orElseGet(this::slowDefault)

**flatMap() - we need to go deeper**

Imagine you have a function that does not accept null but may produce one:

public String findSimilar(@NotNull String s) //...

Using it is a bit cumbersome:

String similarOrNull = x != null? findSimilar(x) : null;

With Optional it is a bit more straighforward:

Optional<String> similar = opt.map(this::findSimilar);

If the function we map() over returns null, the result of map() is an empty Optional. Otherwise it’s the result of said function wrapped with (present) Optional. So far so good but why do we return null-able value if we have Optional?

public Optional<String> tryFindSimilar(String s) //...

Our intentions are clear but using map() fails to produce correct type. Instead we must use flatMap():

Optional<Optional<String>> bad = opt.map(this::tryFindSimilar);

Optional<String> similar = opt.flatMap(this::tryFindSimilar);

Do you see double Optional<Optional<...>>? Definitely not what we want. If you are mapping over a function that returns Optional, use flatMap instead. Here is a simplified implementation of this function:

public <U> Optional<U> flatMap(Function<T, Optional<U>> mapper) {

if (!isPresent())

return empty();

else {

return mapper.apply(value);

}

}

**orElseThrow() - lazily throw exceptions on empty Optional**

Often we would like to throw an exception if value is not available:

public char firstChar(String s) {

if (s != null && !s.isEmpty())

return s.charAt(0);

else

throw new IllegalArgumentException();

}

This whole method can be replaced with the following idiom:

opt.

filter(s -> !s.isEmpty()).

map(s -> s.charAt(0)).

orElseThrow(IllegalArgumentException::new);

We don’t want to create an instance of exception in advance because [creating an exception has significant cost](http://nurkiewicz.blogspot.com/2012/10/where-do-stack-traces-come-from.html).

**Bigger example**

Imagine we have a Person with an Address that has a validFrom date. All of these can be null. We would like to know whether validFrom is set and in the past:

private boolean validAddress(NullPerson person) {

if (person != null) {

if (person.getAddress() != null) {

final Instant validFrom = person.getAddress().getValidFrom();

return validFrom != null && validFrom.isBefore(now());

} else

return false;

} else

return false;

}

Quite ugly and defensive. Alternatively but still ugly:

return person != null &&

person.getAddress() != null &&

person.getAddress().getValidFrom() != null &&

person.getAddress().getValidFrom().isBefore(now());

Now imagine all of these (person, getAddress(), getValidFrom()) are Optionals of appropriate types, clearly indicating they may not be set:

class Person {

private final Optional<Address> address;

public Optional<Address> getAddress() {

return address;

}

//...

}

class Address {

private final Optional<Instant> validFrom;

public Optional<Instant> getValidFrom() {

return validFrom;

}

//...

}

Suddenly the computation is much more streamlined:

return person.

flatMap(Person::getAddress).

flatMap(Address::getValidFrom).

filter(x -> x.before(now())).

isPresent();

Is it more readable? Hard to tell. But at least it’s impossible to produce NullPointerException when Optional is used consistently.

**Converting Optional<T> to List<T>**

I sometimes like to think about Optional as a collection1 having either 0 or 1 elements. This may make understanding of map() and flatMap() easier. Unfortunately Optional doesn’t have toList() method, but it’s easy to implement one:

public static <T> List<T> toList(Optional<T> option) {

return option.

map(Collections::singletonList).

orElse(Collections.emptyList());

}

Or less idiomatically:

public static <T> List<T> toList(Optional<T> option) {

if (option.isPresent())

return Collections.singletonList(option.get());

else

return Collections.emptyList();

}

But why limit ourselves to List<T>? What about Set<T> and other collections? Java 8 already abstracts creating arbitrary collection via [Collectors API](http://download.java.net/jdk8/docs/api/java/util/stream/Collectors.html), introduced for [Streams](http://download.java.net/jdk8/docs/api/java/util/stream/Stream.html). The API is hideous but comprehensible:

public static <R, A, T> R collect(Optional<T> option, Collector<? super T, A, R> collector) {

final A container = collector.supplier().get();

option.ifPresent(v -> collector.accumulator().accept(container, v));

return collector.finisher().apply(container);

}

We can now say:

import static java.util.stream.Collectors.\*;

List<String> list = collect(opt, toList());

Set<String> set = collect(opt, toSet());

In Java 8, I want to do something to an Optional object if it is present, and do another thing if it is not present.

if (opt.isPresent()) {

System.out.println("found");

} else {

System.out.println("Not found");

}

This is not a 'functional style', though.

Optional has an ifPresent() method, but I am unable to chain an orElse() method.

Thus, I cannot write:

opt.ifPresent( x -> System.out.println("found " + x))

.orElse( System.out.println("NOT FOUND"));

In reply to @assylias, I don't think Optional.map() works for the following case:

opt.map( o -> {

System.out.println("while opt is present...");

o.setProperty(xxx);

dao.update(o);

return null;

}).orElseGet( () -> {

System.out.println("create new obj");

dao.save(new obj);

return null;

});

In this case, when opt is present, I update its property and save to the database. When it is not available, I create a new obj and save to the database.

Note in the two lambdas I have to return null.

But when opt is present, both lambdas will be executed. obj will be updated, and a new object will be saved to the database . This is because of the return null in the first lambda. And orElseGet()will continue to execute.