A Pratical Guide to Java 8 Lambda Expressions & Streams

Fu Cheng

A Practical Guide for Java 8 Lambdas and Streams

Mastering Java 8 Lambdas and Streams

Fu Cheng

This book is available at
https://leanpub.com/java8-lambda-expressions-streams

This version was published on 2025-08-09

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2015 - 2025 Fu Cheng

Also By Fu Cheng

Text-to-SQL, Spring AI Implementation with RAG

Understanding Java Virtual Threads

Build AI Applications with Spring AI

From Java 17 to Java 21

Build Native Java Apps with GraalVM

From Java 11 to Java 17

ES6 Generators

Lodash 4 Cookbook

JUnit 5 Cookbook

Contents

1. Introduction **1**

2. Lambda expressions **2**
- 2.1 Start a thread - A simple example 2
- 2.2 Functional interfaces 3
- 2.3 Target typing 4
- 2.4 Lambda expressions 5
- 2.5 Lexical scoping 6
- 2.6 Effectively final local variables 7
- 2.7 Method references 8
  - 2.7.1 Types of method references 9
- 2.8 Default interface methods 9
  - 2.8.1 Static interface methods 11

3. Optional **12**
- 3.1 What's `Optional` 12
- 3.2 Usage of `Optional` 13
  - 3.2.1 Simple usage 13
  - 3.2.2 Chained usage 14
  - 3.2.3 Functional usage 15
- 3.3 How-tos 16
  - 3.3.1 How to interact with legacy library code before `Optional`? 16
  - 3.3.2 How to get value from chained `Optional` reference path? . 17
  - 3.3.3 How to get first value of a list of `Optional`s? 18
  - 3.3.4 How to chain method invocations with return value of `Optional` objects in sequence? 19
  - 3.3.5 How to convert an `Optional` object to a stream? 21
  - 3.3.6 How to use `Optional` to check for `null` and assign default values? 22
- 3.4 Updates after Java 8 22

4. Thank you **24**

1. Introduction

This book is not the first book about Java 8 lambda expressions and streams, and it's definitely not the last one. Java 8 is a Java platform upgrade which the community has been looking forward to for a long time. Lambda expressions and streams quickly gain popularity in Java developers. There are already a lot of books and online tutorials about lambda expressions and streams. This book is trying to explain lambda expressions and streams from a different perspective.

- For lambda expressions, this book explains in details based on JSR 335.
- For streams, this book covers fundamental concepts of Java core library.
- This book provides how-to examples for lambda expressions and streams.
- This book also covers the important utility class `Optional`.

Lambda expressions and streams are easy to understand and use. This book tries to provide some insights about how to use them efficiently.

You can purchase and down PDF/EPUB/MOBI version of this book on Leanpub[1].

[1]https://leanpub.com/java8-lambda-expressions-streams

2. Lambda expressions

When people are talking about Java 8, *lambda expression* is always the first feature to mention. Lambda expression in Java 8 brings functional programming style into Java platform, which has been demanded by Java developers in the community for a long time. Lambda expression is also widely used as a marketing term to promote Java 8 upgrade.

Lambda expressions can increase developers' productivity in many ways. There are a lot of books and online tutorials about Java 8 lambda expressions. This chapter is trying to explain lambda expressions from a different perspective based on the official JSR 335: Lambda Expressions for the Java^{TM} Programming Language[1]. This chapter also provides how-tos for real programming tasks.

2.1 Start a thread - A simple example

Let's start from a simple example which starts a thread and outputs some text to console, see Listing 2.1.

Listing 2.1 Start a thread and output text to console

```
1  public class OldThread {
2    public static void main(String[] args) {
3      new Thread(new Runnable() {
4        public void run() {
5          System.out.println("Hello World!");
6        }
7      }).start();
8    }
9  }
```

The small Java program in Listing 2.1 has 9 lines of code, but only one line of code (line 5) does the real work. All the rest are just boilerplate code. To increase productivity, boilerplate code should be removed as much as possible.

In Listing 2.1, line 1, 2, 8 and 9 are required for Java program to run, so these lines cannot be removed. Line 3 to 7 create a new `java.lang.Thread` object with

[1]https://jcp.org/en/jsr/detail?id=335

an implementation of `java.lang.Runnable` interface, then invoke `Thread` object's `start()` method to start the thread. `Runnable` interface is implemented using an anonymous inner class.

To simplify the code, `new Runnable()` in line 3 and 7 can be removed because the interface type `Runnable` can be deduced from allowed constructors of `Thread` class. Line 4 and 6 can also be removed, because `run()` is the only method in `Runnable` interface.

After removing boilerplate code in line 3, 4, 6 and 7, we get the new code using lambda expressions in Listing 2.2.

Listing 2.2 Lambda expressions style to start a thread

```java
public class LambdaThread {
  public static void main(String[] args) {
    new Thread(() -> System.out.println("Hello World!")).start();
  }
}
```

The new Java program in Listing 2.2 only has 5 lines of code and the core logic is implemented with only one line of code. Lambda expression `() -> System.out.println("Hello World!")` does the same thing as anonymous class in Listing 2.1, but using lambda expression is concise and elegant and easier to understand with less code. Less code means increasing productivity.

Simply put, lambda expression is the syntax sugar to create anonymous inner classes. But there are more to discuss about lambda expressions.

2.2 Functional interfaces

As shown in Listing 2.2, we made two assumptions to remove the boilerplate code. The first assumption is that the interface type, e.g. `Runnable` can be deduced from current context. The second assumption is that there is only one abstract method in the interface. Let's start from the second assumption.

If an interface only has one abstract method (aside from methods of `Object`), it's called *functional interface*. Functional interfaces are special because instances of functional interfaces can be created with *lambda expressions* or *method references*.

In Java SE 7, there are already several functional interfaces, e.g.

- `java.lang.Runnable`

- `java.util.concurrent.Callable`
- `java.util.Comparator`
- `java.io.FileFilter`

Java SE 8 adds a new package `java.util.function` which includes new functional interfaces.

Java developers don't need to care too much about the concept of functional interfaces. If we are trying to use lambda expressions with non-functional interfaces, compiler will throw errors, then we'll know about that. Modern IDEs can also do checks for us. For API and library authors, if an interface is designed to be functional, it should be marked with `@FunctionalInterface` annotation. Compiler will generate an error if the interface doesn't meet the requirements of being a functional interface.

Functional interfaces and single responsibility principle

Single responsibility principle[a] states that every class should only have one responsibility and only one reason to change. Functional interfaces only provide one abstract method to implement, which makes functional interfaces the nature choice to apply single responsibility principle. With the power of lambda expressions, implementing function interfaces is also very elegant. Using functional interfaces is a good combination of design principle and coding practice.

When writing your own application, try to always use functional interfaces. This can also help to create better design.

[a] http://en.wikipedia.org/wiki/Single_responsibility_principle

2.3 Target typing

For the first assumption about lambda expressions, lambda expressions don't have type information. The actual type of a lambda expression is deduced by compiler from its surrounding context at compile time. For example, lambda expression `() -> System.out.println("Hello World!")` can appear in any context where requires an instance of functional interface with the only abstract method takes no arguments and returns `void`. This could be `java.lang.Runnable` interface or any other functional interfaces created by third-party libraries or application code. So the same lambda expression can have different types in different contexts. Expressions like lambda expressions which have deduced type influenced by target type are called *poly expressions*.

Standalone expressions **and** ***Poly expressions***

Standalone expressions are expressions which have type that can be determined from the contents of expressions themselves. For example, `a + b` and `str.substring(1)` are standalone expressions. *Poly expressions* are expressions which have type that can be influenced by target type. Lambda expressions and method references are always poly expressions. Some other expressions may be poly expressions in certain cases.

2.4 Lambda expressions

Lambda expressions have a very simple syntax to write. The basic syntax looks like `(a1, a2) -> {}`. It's similar with a method, which has a list of formal parameters and a body. Lambda expression's syntax is also very flexible.

Listing 2.3 Example of lambda expressions

```
 1  list -> list.size()
 2
 3  (x, y) -> x * y
 4
 5  (double x, double y) -> x + y
 6
 7  () -> "Hello World"
 8
 9  (String operator, double v1, double v2) -> {
10    switch (operator) {
11      case "+":
12        return v1 + v2;
13      case "-":
14        return v1 - v2;
15      default:
16        return 0;
17    }
18  }
```

In Listing 2.3, lambda expression in line 1 only has one formal parameter, so parentheses are omitted. Expression's body is a single expression, so braces are also omitted. Formal parameter `list` has no type declaration, so the type of `list` is implicitly inferred by compiler. Lambda expression in line 3 has two formal parameters, so parentheses around parameters list are required. Lambda expression in line 5 has explicit type for formal parameters `x` and `y`. Lambda expression in line 7 has no formal parameters. In this case, parentheses are required. Lambda expression starts from line 9 is a complicated example with three formal parameters and multiple lines of code in body. In this case, braces are required to wrap the body.

Lambda expression's body can return a value or nothing. If a value is returned, the type of return value must be compatible with target type. If an exception is thrown by the body, the exception must be allowed by target type's `throws` declaration.

2.5 Lexical scoping

In the body of lambda expressions, it's a common requirement to access variables in the enclosing context. Lambda expression uses a very simple approach to handle resolution of names in the body. Lambda expressions don't introduce a new level of scoping. They are lexically scoped, which means names in lambda expression's body are interpreted as they are in the expression's enclosing context. So lambda expression body is executed as it's in the same context of code enclosing it, except that the body can also access expression's formal parameters. `this` used in the expression body has the same meaning as in the enclosing code.

Listing 2.4 Lexical scoping of lambda expression

```java
public void run() {
  String name = "Alex";
  new Thread(() -> System.out.println("Hello, " + name)).start();
}
```

In Listing 2.4, lambda expression uses variable `name` from enclosing context. Listing 2.5 shows the usage of `this` in lambda expression body. `this` is used to access `sayHello` method in the enclosing context.

Listing 2.5 `this` in lambda expression

```
public class LambdaThis {
  private String name = "Alex";

  public void sayHello() {
    System.out.println("Hello, " + name);
  }

  public void run() {
    new Thread(() -> this.sayHello()).start();
  }

  public static void main(String[] args) {
    new LambdaThis().run();
  }
}
```

From code examples in Listing 2.4 and Listing 2.5, we can see how lambda expressions make developers' life much easier by simplifying the scope and name resolution.

2.6 Effectively final local variables

To capture variables from enclosing context in lambda expression body, the variables must be `final` or **effectively final**. `final` variables are declared with `final` modifiers. **effectively final** variables are never assigned after their initialization. Code in Listing 2.6 has compilation error, because variable `name` is assigned again after its initialization, so `name` cannot be referenced in lambda expression body.

Listing 2.6 Compilation error of using non-final variables in lambda expression body

```
public void run() {
  String name = "Alex";
  new Thread(() -> System.out.println("Hello, " + name)).start();
  name = "Bob";
}
```

2.7 Method references

Method references are references to existing methods by name. For example, `Object::toString` references `toString` method of `java.lang.Object` class. `::` is used to separated class or instance name and method name. Method references are similar with lambda expressions, except that method references don't have a body, just refer existing methods by name. Method references can be used to remove more boilerplate code. If the body of a lambda expression only contains one line of code to invoke a method, it can be replaced with a method reference.

`StringProducer` in Listing 2.7 is a functional interface with method `produce`.

Listing 2.7 Functional interface to produce a string

```
@FunctionalInterface
public interface StringProducer {
  String produce();
}
```

Listing 2.8 is an example of using `StringProducer`. `displayString` method takes an instance of `StringProducer` and outputs the string to console. In `run` method, method reference `this::toString` is used to create an instance of `StringProducer`. Invoking `run` method will output text like `StringProducerMain@65ab7765` to the console. It's the same result as invoking `toString()` method of current `StringProducerMain` object. Using `this::toString` is the same as using lambda expression `() -> this.toString()`, but in a more concise way.

Listing 2.8 Usage of `StringProducer` with method reference

```
public class StringProducerMain {
  public static void main(String[] args) {
    new StringProducerMain().run();
  }

  public void run() {
    displayString(this::toString);
  }

  public void displayString(StringProducer producer) {
    System.out.println(producer.produce());
  }
}
```

2.7.1 Types of method references

There are different types of method references depends on the type of methods they refer to.

Static method
: Refer to a static method using `ClassName::methodName`, e.g. `String::format`, `Integer::max`.

Instance method of a particular object
: Refer to an instance method of a particular object using `instanceRef::methodName`, e.g. `obj::toString`, `str::toUpperCase`.

Method from superclass
: Refer to an instance method of a particular object's superclass using `super::methodName`, e.g. `super::toString`. In Listing 2.8, if replacing `this::toString` with `super::toString`, then `toString` of `Object` class will be invoked instead of `toString` of `StringProducerMain` class.

Instance method of an arbitrary object of a particular type
: Refer to an instance method of any object of a particular type using `ClassName::methodName` e.g. `String::toUpperCase`. The syntax is the same as referring a static method. The difference is that the first parameter of functional interface's method is the invocation's receiver.

Class constructor
: Refer to a class constructor using `ClassName::new`, e.g. `Date::new`.

Array constructor
: Refer to an array constructor using `TypeName[]::new`, e.g. `int[]::new`, `String[]::new`.

2.8 Default interface methods

The introduction of *default methods* is trying to solve a longstanding problem in Java interface design: how to evolve a published interface. Java's interface is a very tight contract between an API and its client. Before Java 8, once an interface is published and implemented by other clients, it's very hard to add a new method to the interface without breaking existing implementations. All implementations must be updated to implement the new method. This means interface design has to be done correctly at the first time and cannot happen iteratively without breaking existing code.

Before Java 8, an interface can only have *abstract* methods. Abstract methods must be implemented by implementation classes. Default methods of interfaces in Java 8 can have both declarations and implementations. Implementation of a default method will be inherited by classes which don't override it. If we want to add a new method to an existing interface, this new method can have default implementation. Then existing code won't break because this new method will use the default implementation. New code can override this default implementation to provide a better solution. In the default method's implementation, only existing interface methods can be used.

Listing 2.9 is a simple example about how to use default interface methods. The scenario is to insert records into a database.

Listing 2.9 First version of interface `RecordInserter` to insert records

```java
public interface RecordInserter {
  void insert(Record record);
}
```

In the first version, interface `RecordInserter` only has one abstract method `insert`. `SimpleRecordInserter` class in Listing 2.10 is the first implementation.

Listing 2.10 First implementation of `RecordInserter` interface

```java
public class SimpleRecordInserter implements RecordInserter {
  @Override
  public void insert(Record record) {
    System.out.println("Inserting " + record);
  }
}
```

Then we find out that the performance of inserting a lot of records is not very good, so we want to add batch inserting to improve performance. So a new method `insertBatch` is added to the `RecordInserter` interface with default implementation. `insertBatch` takes a list of `Record` as input, so the default implementation just iterates the list and uses existing `insert` method to insert `Record` one by one. This default implementation can make sure `SimpleRecordInserter` class still works.

Listing 2.11 `RecordInserter` interface with batch insert

```java
public interface RecordInserter {
  void insert(Record record);

  default void insertBatch(List<Record> recordList) {
    recordList.forEach(this::insert);
  }
}
```

To improve the performance, a new implementation `FastRecordInserter` class overrides the default implementation of `insertBatch` method and create a better solution.

Listing 2.12 New implementation of `insertBatch` method

```java
public class FastRecordInserter extends SimpleRecordInserter {
  @Override
  public void insertBatch(List<Record> recordList) {
    System.out.println("Inserting " + recordList);
    //Use SQL batch operations to insert
  }
}
```

2.8.1 Static interface methods

In Java 8, interfaces can also have static methods. Helper methods related to an interface can be added as static methods, see Listing 2.13.

Listing 2.13 Interface with a static method

```java
public interface WithStaticMethod {
  static void simpleMethod() {
    System.out.println("Hello World!");
  }
}
```

3. Optional

`NullPointerException`s may be the most seen exceptions during Java developers' daily development. Null reference is considered as **The billion dollar mistake** by its inventor Tony Hoare[1]. Maybe `null` should not be introduced into Java language in the first place. But it's already there, so we have to live with it.

Suppose we have a method with argument `val` of a non-primitive type, we should check first to make sure the value of `val` is not `null`.

Listing 5.1. Check `null` value

```
public void myMethod(Object val) {
  if (val == null) {
    throw new IllegalArgumentException("val cannot be null!");
  }
  // Use of val
}
```

Listing 5.1 shows a common pattern of writing code that takes arguments with possible `null` values. If a method accepts multiple arguments, all these arguments should be checked. Utility methods like `Objects.requireNonNull()` can help, but the code is still long and tedious. Another case is for long object references chains, e.g. `a.b.c`, then all objects in the reference chain need to be checked. Groovy has safe navigation operator[2] `?.`, but Java doesn't the same thing.

3.1 What's `Optional`

Java 8 introduced a new class `Optional<T>`[3] to solve the issues with `null`s. The idea behind `Optional` is quite simple. An `Optional` object is a holder of the actual object which may be `null`. Client code interacts with the `Optional` object instead of the actual object. The `Optional` object can be queried about the existence of the actual object. Although the actual object may be `null`, there is always an non-null `Optional` object.

[1]http://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare
[2]http://docs.groovy-lang.org/latest/html/documentation/#_safe_navigation_operator
[3]https://docs.oracle.com/javase/8/docs/api/java/util/Optional.html

The major benefit of using `Optional` is to force client code to deal with the situation that the actual object that it wants to use may be `null`. Instead of using the object reference directly, the `Optional` object needs to be queried first.

`Optional` objects are very easy to create. If we are sure the actual object is not `null`, we can use `<T> Optional<T> of(T value)` method to create an `Optional` object, otherwise we should use `<T> Optional<T> ofNullable(T value)` method. If we just want to create an `Optional` object which holds nothing, we can use `<T> Optional<T> empty()`.

Listing 5.2. Create `Optional` objects

```
Optional<String> opt1 = Optional.of("Hello");

Optional<String> opt2 = Optional.ofNullable(str);

Optional<String> opt3 = Optional.empty();
```

If you pass a `null` value to `Optional.of()` method, it throws a `NullPointerException`.

3.2 Usage of `Optional`

3.2.1 Simple usage

The simple usage of `Optional` is to query it first, then retrieve the actual object hold by it. We can use `boolean isPresent()` method to check if an `Optional` object holds a non-null object, then use `T get()` method to get the actual value.

Listing 5.3. Simple usage of `Optional`

```
public void myMethod(Optional<Object> holder) {
  if (!holder.isPresent()) {
    throw new IllegalArgumentException("val cannot be null!");
  }
  Object val = holder.get();
}
```

Listing 5.3 is similar with Listing 5.1, except it uses `Optional`. But the code in Listing 5.3 is still long and tedious.

3.2.2 Chained usage

Since `Optionals` are commonly used with `if..else`, `Optional` class has built-in support for this kind of scenarios.

`void ifPresent(Consumer<? super T> consumer)` method invokes the specified consumer function when value is present, otherwise it does nothing.

Listing 5.4 Example of `Optional.ifPresent`

```java
public void output(Optional<String> opt) {
  opt.ifPresent(System.out::println);
}
```

`T orElse(T other)` returns the value if it's present, otherwise return the specified `other` object. This `other` object acts as the default or fallback value.

Listing 5.5. Example of `Optional.orElse`

```java
public String getHost(Optional<String> opt) {
  return opt.orElse("localhost");
}
```

`T orElseGet(Supplier<? extends T> other)` is similar with `orElse()`, except a supplier function is invoked to get the value if not present. `getPort()` method in Listing 5.6 invokes `getNextAvailablePort()` method to get the port if no value is present.

Listing 5.6 Example of `Optional.orElseGet`

```java
public int getNextAvailablePort() {
  int min = 49152;
  int max = 65535;
  return new Random().nextInt((max - min) + 1) + min;
}

public int getPort(Optional<Integer> opt) {
  return opt.orElseGet(this::getNextAvailablePort);
}
```

`<X extends Throwable> T orElseThrow(Supplier<? extends X> exceptionSupplier)` returns the value if present, otherwise invokes the supplier function to get the exception to throw.

Listing 5.7 Example of `Optional.orElseThrow`

```
public void anotherMethod(Optional<Object> opt) {
  Object val = opt.orElseThrow(IllegalArgumentException::new);
}
```

3.2.3 Functional usage

As `Optional` objects are the holders of actual objects, it's inconvenient to use when manipulating the actual objects. `Optional` provides some methods to use it in a functional way.

`Optional<T> filter(Predicate<? super T> predicate)` filters an `Optional` object's value using the specified predicate. If the `Optional` object holds a value and the value matches the predicate, an `Optional` object with the value is returned, otherwise an empty `Optional` object is returned.

Listing 5.8 only outputs a string when it is not empty.

Listing 5.8 Example of `Optional.filter`

```
Optional<String> opt = Optional.of("Hello");
opt.filter((str) -> str.length() > 0).ifPresent(System.out::println);
```

`<U> Optional<U> map(Function<? super T,? extends U> mapper)` applies the specified mapping function to the `Optional`'s value if present. If the mapping result is not `null`, it returns an `Optional` object with this mapping result, otherwise returns an empty `Optional` object.

Listing 5.9 outputs the length of input string.

Listing 5.9 Example of `Optional.map`

```
Optional<String> opt = Optional.of("Hello");
opt.map(String::length).ifPresent(System.out::println);
```

`<U> Optional<U> flatMap(Function<? super T,Optional<U>> mapper)` is similar with `map()`, but the mapping function returns an `Optional` object. The returned `Optional` object only contains a value when the value is present in both the current `Optional` object and the mapped `Optional` object.

`flatMap()` is very useful when handling long reference chain. For example, considering a `Customer` class has a method `getAddress()` to return an `Optional<Address>`

object. The `Address` class has a method `getZipCode()` to return an `Optional<ZipCode>` object. Listing 5.10 uses two chained `flatMap()` method invocations to get an `Optional<ZipCode>` object from an `Optional<Customer>` object. With the use of `flatMap()`, it's very easy to create safe chained object references.

Listing 5.10 Example of `Optional.flatMap`

```
Optional<Customer> opt = getCustomer();
Optional<ZipCode> zipCodeOptional = opt.flatMap(Customer::getAddress)
  .flatMap(Address::getZipCode);
zipCodeOptional.ifPresent(System.out::println);
```

`Optional` class is primarily designed to be used as holder objects in return values of methods. You shouldn't use `Optional` as the type of method parameters or fields.

3.3 How-tos

3.3.1 How to interact with legacy library code before `Optional`?

Not all library code has been updated to use Java 8 `Optional`, so when we use third-party libraries, we need to wrap an object using `Optional`.

Listing 5.11 Old legacy library code

```
public Date getUpdated() {
  //Get date from somewhere
}

public void setUpdated(Date date) {

}
```

Listing 5.12 Wrap legacy library code with `Optional`

```
public Optional<Date> getUpdated() {
  return Optional.ofNullable(obj.getUpdate());
}
```

3.3.2 How to get value from chained `Optional` reference path?

Given an object reference path `a.getB().getC()` with `getB()` and `getC()` methods both return `Optional` objects, we can use `flatMap()` method to get the actual value. In Listing 5.13, we use `flatMap()` to get value of reference path `a.b.c.value`.

Listing 5.13 Get value from chained `Optional` reference path

```
package optional;

import java.util.Optional;

public class OptionalReferencePath {

  public static void main(String[] args) {
    new OptionalReferencePath().test();
  }

  public void test() {
    A a = new A();
    String value = a.getB()
        .flatMap(B::getC)
        .flatMap(C::getValue)
        .orElse("Default");
    System.out.println(value);
  }

  static class C {

    private final String value = "Hello";

    public Optional<String> getValue() {
      return Optional.ofNullable(value);
    }
  }
  static class B {

    private final C c = new C();

    public Optional<C> getC() {
      return Optional.ofNullable(c);
    }
  }

  static class A {

    private final B b = new B();

    public Optional<B> getB() {
      return Optional.ofNullable(b);
    }
  }
}
```

3.3.3 How to get first value of a list of `Optional`s?

Given a list of `Optional` objects, we can use code in Listing 5.14 to get the first value.

Listing 5.14 Get first value of a list of `Optional`s

```
package optional;

import java.util.Arrays;
import java.util.List;
import java.util.Optional;

public class ListOfOptionals {

  public static void main(String[] args) {
    List<Optional<String>> optionalList = Arrays.asList(
        Optional.empty(),
        Optional.of("hello"),
        Optional.ofNullable(null),
        Optional.of("world")
    );
    String value = optionalList.stream()
```

```
        .filter(Optional::isPresent)
        .map(Optional::get)
        .findFirst()
        .orElse(null);
    System.out.println(value);
  }
}
```

Starting from Java 9, you can use `optionalList.stream().flatMap(Optional::stream)` to convert an object of type `Stream<Optional<T>>` to type `Stream<T>`.

3.3.4 How to chain method invocations with return value of `Optional` objects in sequence?

Suppose we want to retrieve a value to be used in the client code and we have different types of retrievers to use, the code logic is that we try the first retriever firstly, then the second retriever and so on, until a value is retrieved. In Listing 5.15, `ValueRetriever` is the interface of retrievers with only one method `retrieve()` which returns an `Optional<Value>` object. Classes `ValueRetriever1`, `ValueRetriever2` and `ValueRetriever3` are different implementations of the interface `ValueRetriever`.

Listing 5.15 shows two different ways to chain method invocations. The first method `retrieveResultOrElseGet()` uses `Optional`'s `orElseGet()` method to invoke a `Supplier` function to retrieve the value. The second method `retrieveResultStream()` uses `Stream.of()` to create a stream of `Supplier<Optional<Value>>` objects, then filters the stream to only include `Optional`s with values, then gets the first value.

Listing 5.15 Chain method invocations with return value of `Optional` objects in sequence

```
package optional;

import java.util.Optional;
import java.util.function.Supplier;
import java.util.stream.Stream;

public class ChainedOptionals {

  private final ValueRetriever retriever1 = new ValueRetriever1();
  private final ValueRetriever retriever2 = new ValueRetriever2();
  private final ValueRetriever retriever3 = new ValueRetriever3();

  public Value retrieveResultOrElseGet() {
    return retriever1.retrieve()
        .orElseGet(() -> retriever2.retrieve()
            .orElseGet(() -> retriever3.retrieve().orElse(null)));
  }

  public Value retrieveResultStream() {
    return Stream.<Supplier<Optional<Value>>>of(
            retriever1::retrieve,
            retriever2::retrieve,
            retriever3::retrieve)
        .map(Supplier::get)
        .filter(Optional::isPresent)
        .map(Optional::get)
        .findFirst()
        .orElse(null);
  }

  public static void main(String[] args) {
    ChainedOptionals chainedOptionals = new ChainedOptionals();
    Value value = chainedOptionals.retrieveResultOrElseGet();
    System.out.println(value);
    value = chainedOptionals.retrieveResultStream();
    System.out.println(value);
  }

  static class Value {

    private final String value;

    public Value(String value) {
      this.value = value;
    }

    @Override
    public String toString() {
      return String.format("Value -> [%s]", this.value);
    }
  }

  interface ValueRetriever {

    Optional<Value> retrieve();
  }
  static class ValueRetriever1 implements ValueRetriever {

    @Override
    public Optional<Value> retrieve() {
      return Optional.empty();
    }
  }

  static class ValueRetriever2 implements ValueRetriever {

    @Override
    public Optional<Value> retrieve() {
      return Optional.of(new Value("hello"));
    }
  }

  static class ValueRetriever3 implements ValueRetriever {

    @Override
    public Optional<Value> retrieve() {
      return Optional.of(new Value("world"));
    }
  }
}
```

The output of the program is two lines of `Value -> [hello]`, because `ValueRetriever2` is the first retriever that returns a value.

As streams are lazily evaluated, `Supplier` functions in Listing 5.15 won't be invoked unless the last `Supplier` function returns an empty `Optional` object. This can make sure that there won't be any unnecessary method invocations.

3.3.5 How to convert an `Optional` object to a stream?

Sometimes we may want to convert an `Optional` object to a stream to work with `flatMap`. `optionalToStream` method in Listing 5.16 converts an `Optional` object to a stream with a single element or an empty stream.

Listing 5.16 Convert an `Optional` object to a stream

```
public <T> Stream<T> optionalToStream(Optional<T> opt) {
  return opt.isPresent() ? Stream.of(opt.get()) : Stream.empty();
}
```

Starting from Java 9, you can use the `Optional.stream()` method in Java standard library.

3.3.6 How to use `Optional` to check for `null` and assign default values?

It's a common case to check if a value is `null` and assign a default value to it if it's `null`. `Optional` can be used to write elegant code in this case. Listing 5.17 shows the traditional way to check for `null` which uses four lines of code.

Listing 5.17 Traditional way to check for `null`

```
public void test() {
  MyObject myObject = calculate();
  if (myObject == null) {
    myObject = new MyObject();
  }
}
```

Listing 5.18 shows how to use `Optional` to do the same thing as Listing 5.17with only one line of code.

Listing 5.18 Use `Optional` to simplify code

```
public void test() {
  MyObject myObject = Optional.ofNullable(calculate()).orElse(new MyObject());
}
```

3.4 Updates after Java 8

After Java 8 was released, new methods have been added to `Optional`.

`isEmpty()` method, introduced in Java 11, is the opposite of `isPresent()`.

`ifPresentOrElse(Consumer<? super T> action, Runnable emptyAction)` method, introduced in Java 9, performs the given action when the value is present, otherwise performs the `emptyAction`.

`or(Supplier<? extends Optional<? extends T>> supplier)` method, introduced in Java 9, returns the current `Optional` object if the value is present, otherwise invokes the supplier function to return the `Optional` object.

4. Thank you

Thank you for reading the sample chapters of this book. You can purchase this book on Leanpub[1].

[1] https://leanpub.com/java8-lambda-expressions-streams