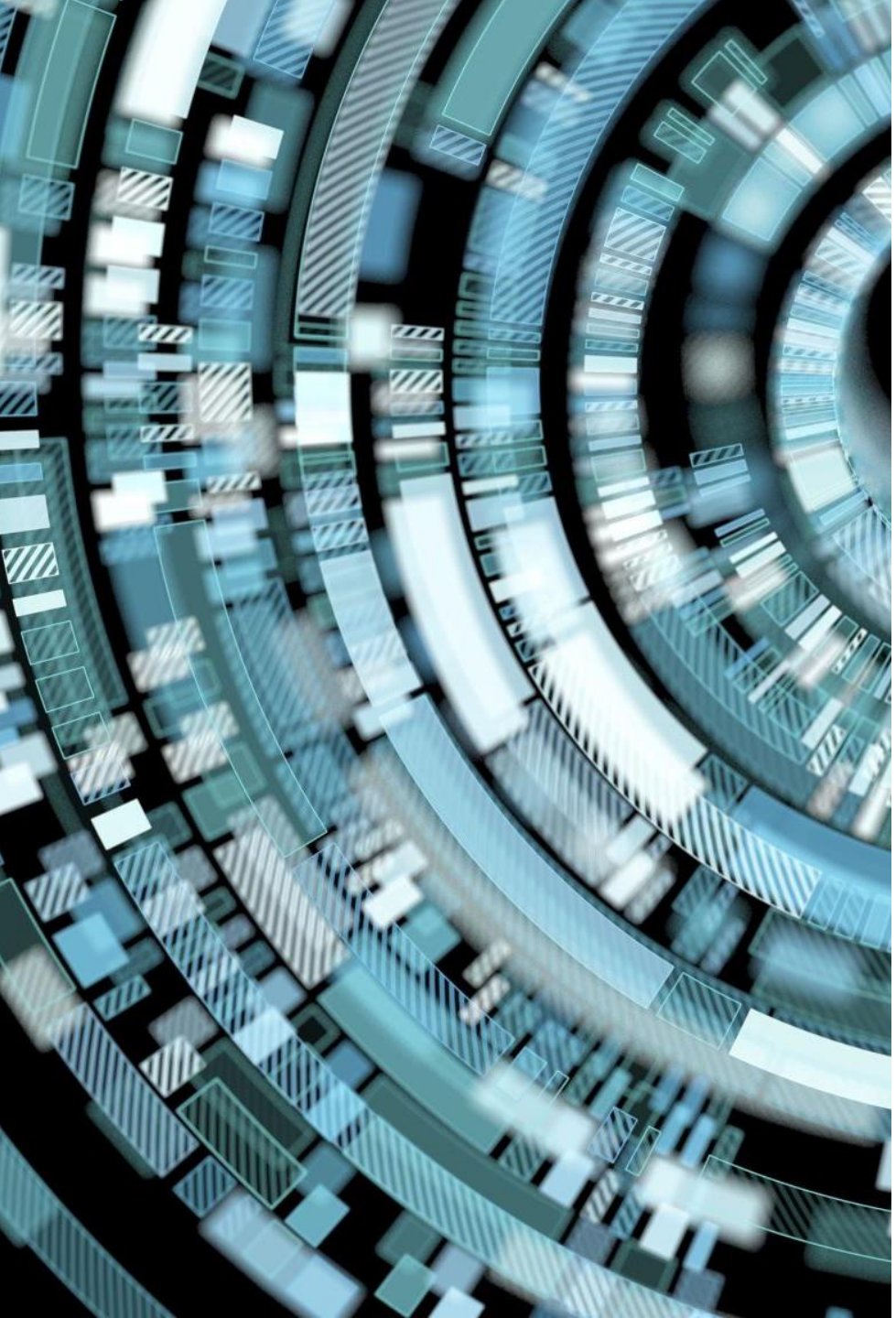
Computer System Design & Application

计算机系统设计与应用A

陶伊达 （TAO Yida）
taoyd@sustech.edu.cn



Lecture 4

- Streams API
- Optional<T>

Java Stream API Overview

```
public interface Stream<T>
```

- The `Stream` API is introduced in Java 8 (`java.util.stream`), don't confuse it with I/O Streams!

- Used to process collections of objects
  - Data stream is obtained from a source
  - Data stream is processed through chained intermediate operations (pipeline)
  - Getting the result from a terminal operation

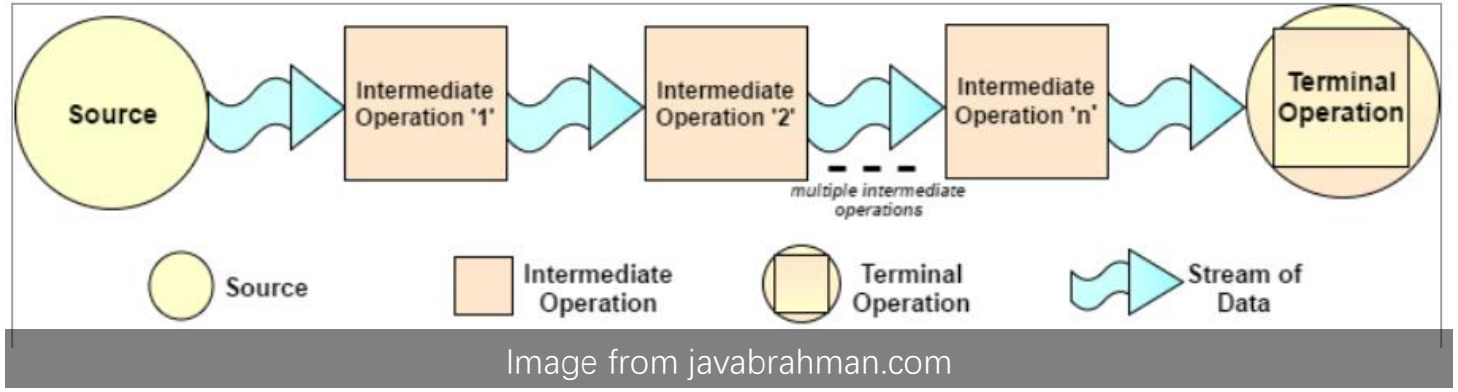


Image from javabrahman.com

Create a Stream

```
default Stream<E> stream()
```

Returns a sequential `Stream` with this collection as its source.

- Approach I: getting a `Stream` from a Java `Collection`, which has the `stream()` method

```
List<String> list = new ArrayList<String>();

list.add(“a”);
list.add(“b”);

Stream<String> stream = list.stream();
```

Create a Stream

```
static <T> Stream<T> generate(Supplier<T> s)
```

- Approach II: using `Stream.generate()`, which needs a `Supplier` as input

```
Stream<String> echos = Stream.generate(() -> "Echo");

Stream<Double> randoms = Stream.generate(Math::random);
```

Could generate "infinite" streams

Create a Stream

```
static <T> Stream<T> generate(Supplier<T> s)
```

- Approach II: using `Stream.generate()`, which needs a `Supplier` as input

Example: generate a stream of natural numbers and print the first 20

```
Stream<Integer> natual = Stream.generate(new NatualSupplier());
natual.limit(20).forEach(System.out::println);


class NatualSupplier implements Supplier<Integer> {
    int n = 0;
    public Integer get() {
        n++;
        return n;
    }
}
```

Example from
https://www.liaoxuefeng.com/wiki/1252599548343744/1322655160467490

Create a Stream

```
static <T> Stream<T> of(T... values)
```

- Approach III: using `Stream.of()`, which has a varargs paramter (take any number of arguments)

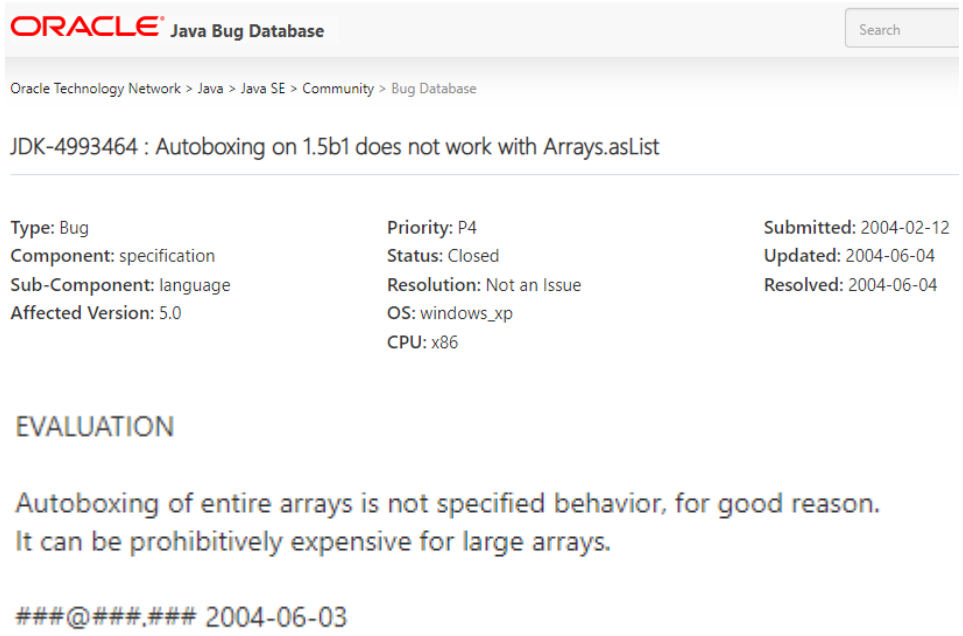
```
Integer[] array = new Integer[]{1,2,3};
Stream<Integer> istream = Stream.of(array);
```

```
Stream<String> sentence = Stream.of("This","is","Java","2");
```

What about int[]?

Working with `int[]`

How to convert `int[]` to `List<Integer>`?



ORACLE Java Bug Database

Search

Oracle Technology Network > Java > Java SE > Community > Bug Database

JDK-4993464 : Autoboxing on 1.5b1 does not work with Arrays.asList

Type: Bug
Component: specification
Sub-Component: language
Affected Version: 5.0

Priority: P4
Status: Closed
Resolution: Not an Issue
OS: windows_xp
CPU: x86

Submitted: 2004-02-12
Updated: 2004-06-04
Resolved: 2004-06-04

EVALUATION

Autoboxing of entire arrays is not specified behavior, for good reason.
It can be prohibitively expensive for large arrays.

###@###.### 2004-06-03

```
int[] ints = new int[]{1,2,3,4,5};
List<Integer> list1 = Arrays.stream(ints).boxed().toList();
List<Integer> list2 = IntStream.of(ints).boxed().toList();
```

https://bugs.java.com/bugdatabase/view_bug.do?bug_id=4993464

Primitive Type Streams

- The stream library has specialized types `IntStream`, `LongStream`, and `DoubleStream` that store primitive values directly, without using wrappers (e.g., Integer).

```
IntStream stream0 = Arrays.stream(new int[]{1,2,3});

IntStream stream1 = IntStream.of(1,2,3,5,8);

IntStream stream2 = IntStream.range(5,10);

Stream<String> sentences = Stream.of("This","is","Java","2");
IntStream stream3 = sentences.mapToInt(String::length);
```

Intermediate Operations

- Intermediate (non-terminal) operations transform or filter the elements in the stream
  - `filter()`
  - `map()`
  - `sorted()`
  - `distinct()`
  - `peek(), limit(), skip()`
- We get a new stream back as the result when adding an intermediate operation to a stream
- [Lazy evaluation] All intermediate operations do not get executed until a terminal operation is invoked (discussed later)

filter()

```
Stream<T> filter(Predicate<? super T> predicate)
```

```
Predicate<T>          boolean test(T t)
```

The `Predicate` interface represents functions who take an argument and return a boolean

- Returns a stream consisting of the elements of this stream that match the given predicate.

```
List<Integer> list = Arrays.asList(10,20,33,43,54,68);
list.stream()
    .filter(element -> (element % 2==0))
    .forEach(element -> System.out.print(element+ " "));
```

map()

```
<R> Stream<R> map(Function<? super T,? extends R> mapper)
```

```
Function<T,R>        R apply(T t)
```

The `Function` interface represents functions whose result and argument types could differ

- Returns a stream consisting of the results of applying/mapping the given function to the elements of this stream.

```
List<String> strList = new ArrayList<String>();
strList.add("123");
strList.add("456");

strList.stream()          Stream<String>
        .map(Integer::parseInt)   Stream<Integer>
        .forEach(System.out::println);
```

distinct()

- Returns a stream consisting of the distinct elements (removing duplicates and keeping only one of them)

```
List<String> strList = new ArrayList<String>();
strList.add("apple");
strList.add("orange");
strList.add("banana");
strList.add("apple");

List<String> result = strList.stream()
                .distinct()
                .collect(Collectors.toList());
```

sorted()

- `sorted()`: sort the elements by natural order
  `list.stream().sorted().forEach(System.out::println)`
- `sorted(Comparator<? super T> comparator)`: sort the elements according to the given `Comparator`

```
class Point
{
        Integer x, y;
        Point(Integer x, Integer y) {
                this.x = x;
                this.y = y;
        }

}
```

Point's toString() is omitted here

```
aList.stream()
      .sorted((p1, p2)->p1.x.compareTo(p2.x))
      .forEach(System.out::println);
```

Example: https://www.geeksforgeeks.org/stream-sorted-in-java/

Stream Pipeline/Chain Example

- A stream pipeline consists of a stream source, followed by zero or more intermediate operations, and a terminal operation.

```
List<Integer> ilist = Arrays.asList(1,2,3,4,5,6,7,8,9);
ilist.stream()
        .filter(element -> (element % 2 == 1))
        .limit(3)
        .map(element -> (element*element))
        .forEach(System.out::println);
```

Terminal Operation

```
anyMatch()
allMatch()
noneMatch()
collect()
count()
findAny()
findFirst()
forEach()
min()
max()
reduce()
toArray()
```

- A terminal operation marks the end of the stream and is always the last operation in the stream pipeline
- A terminal operation returns a non-stream type of result
  - Return primitive type (`count()`)
  - Return reference type (`collect()`)
  - Return void (`forEach()`)
- [Eager execution] Terminal operations are early executed (later)

anyMatch()

`Predicate<T>`

The `Predicate` interface represents functions who take an argument and return a boolean

`boolean test(T t)`

```
boolean anyMatch(Predicate<? super T> predicate)
```

- Returns whether any elements of this stream match the provided predicate (check whether any element in list satisfies a given condition)

```
boolean x = sList.stream().anyMatch((value) -> { return value.startsWith("Java"); });

boolean x = sList.stream().anyMatch(str -> Character.isUpperCase(str.charAt(1)));
```

findFirst()

- Returns an `Optional` describing the first element of this stream, or an empty `Optional` if the stream is empty

```
List<String> stringList = new ArrayList<String>();

stringList.add("one");
stringList.add("two");
stringList.add("three");

Stream<String> stream = stringList.stream();
Optional<String> result = stream.findFirst();

System.out.println(result.orElse("unknown"));
```

Collecting Results

```
Stream<String> stream = Stream.of("a", "bb", "cc", "ddd");
```

- When you are done with a stream, you often want to collect the result in a data structure

```
String[] result = stream.toArray(String[]::new)
```

- The `Collectors` class provides a set of factory methods for common collectors

```
List<String> result = stream.collect(Collectors.toList())

Set<String> result = stream.collect(Collectors.toSet())

TreeSet<String> result = stream.collect(Collectors.toCollection(TreeSet::new))
```

Collecting Results

```
Stream<String> stream = Stream.of("a", "bb", "cc", "ddd");
```

```
Map<String, Integer> map =
stream.collect(Collectors.toMap(Function.identity(), String::length));
```

{bb=2, cc=2, a=1, ddd=3}

```
String joined =
stream.collect(Collectors.joining("$"));
```

abbcc$ddd

Collecting Results

We use `Collectors.groupingBy` for grouping objects by some property and storing results in a `Map` instance.

```
Stream<String> stream = Stream.of("a", "bb", "cc", "ddd", "a", "bb");
Map<String, Long> group =
stream.collect(Collectors.groupingBy(Function.identity(), Collectors.counting()));
```

```
Stream<String> stream = Stream.of("a", "bb", "cc", "ddd", "a", "bb", "eee");
Map<Integer, Set<String>> group =
stream.collect(Collectors.groupingBy(String::length, Collectors.toSet()));
```

Collecting Results

We use **`Collectors.groupingBy`** for grouping objects by some property and storing results in a **`Map`** instance.

```
Stream<String> stream = Stream.of("1a", "1bb", "1c", "2a", "2a", "2bb");

Map<Character, Set<String>> group =
                        stream.collect(Collectors.groupingBy(s->s.charAt(0),
                         Collectors.mapping(s->s.substring(1), Collectors.toSet())));
```

Reduction

- A reduction is a terminal operation that aggregates a stream into a type or a primitive
- Reduction operations in Java Stream API
  - `min()`
  - `max()`
  - `average()`
  - `sum()`
  - `reduce(): <- the general one`

reduce()

```
BinaryOperator<T>    T apply(T t1, T t2)
```

```
Optional<T> reduce(BinaryOperator<T> accumulator)
```

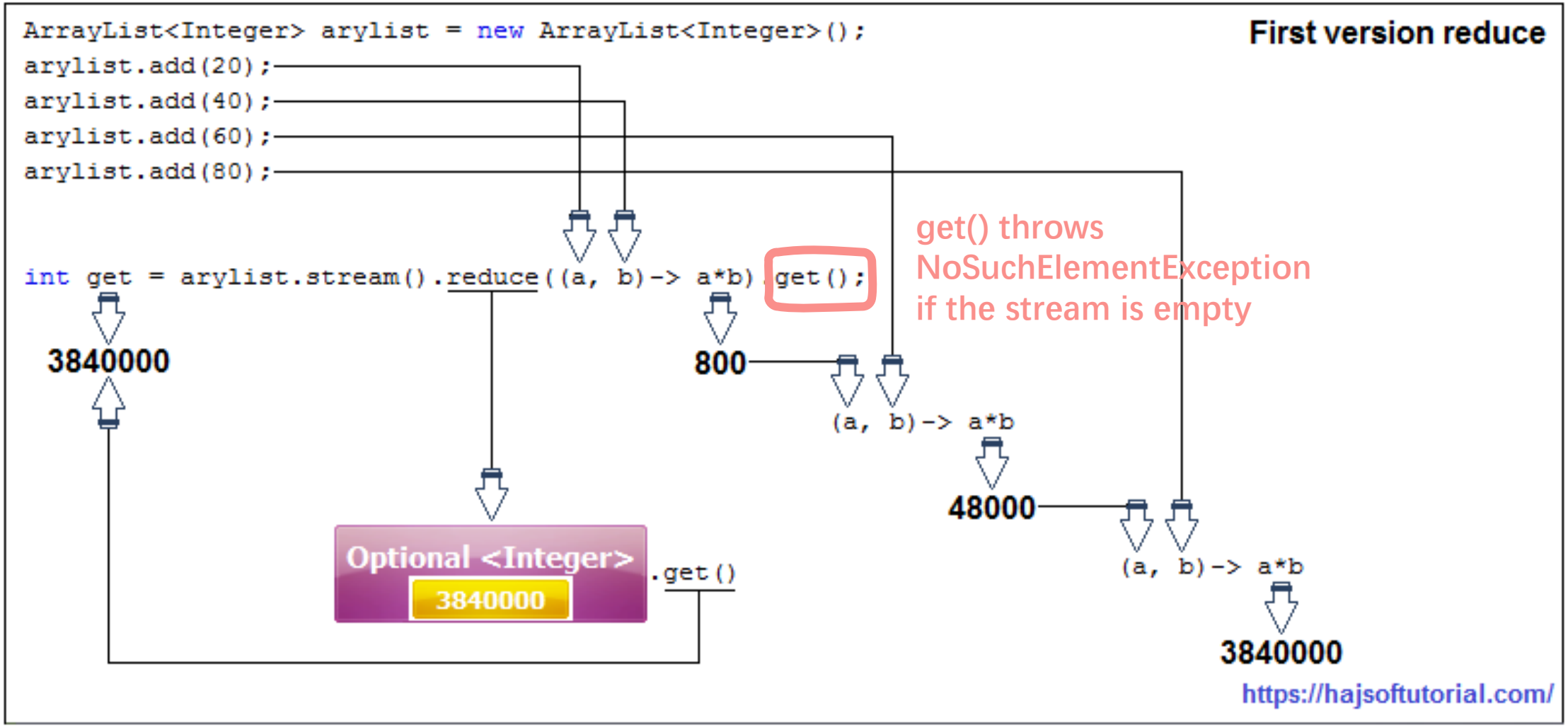
Next element: e.g., the next number in the stream

```
reduce( (a, b) -> a + b )
```

Partial result: e.g., the sum of all processed numbers so far

Accumulator function: e.g., add two numbers

reduce() Example I



reduce() Example I

- To avoid the exception potentially thrown by `get()`, you may use:

```
int sum = arrayList.stream().reduce((a,b)->a+b).orElse(0);

int sum = arrayList.stream().reduce(0, (a,b)->a+b);
```

The identity element is both the initial value of the reduction and the default result if there are no elements in the stream

reduce() Example II

What's the output?

```
List<String> words = Arrays.asList("Java", "Python", "C", "C++", "JavaScript");

String x = words.stream()
                .reduce((word1, word2) -> word1.length() > word2.length() ? word1 : word2)
                .orElse("");

System.out.println(x);
```

Lazy Evaluation

Benefits?

Intermediate operations are lazily executed

- They only remember the operations, but don't do anything right away (lazy)

Terminal operations are eagerly executed

- When terminal operations are initiated, the remembered operations are performed one by one (eager)

Example

How many operations do we have to perform when the code is executed eagerly / lazily?

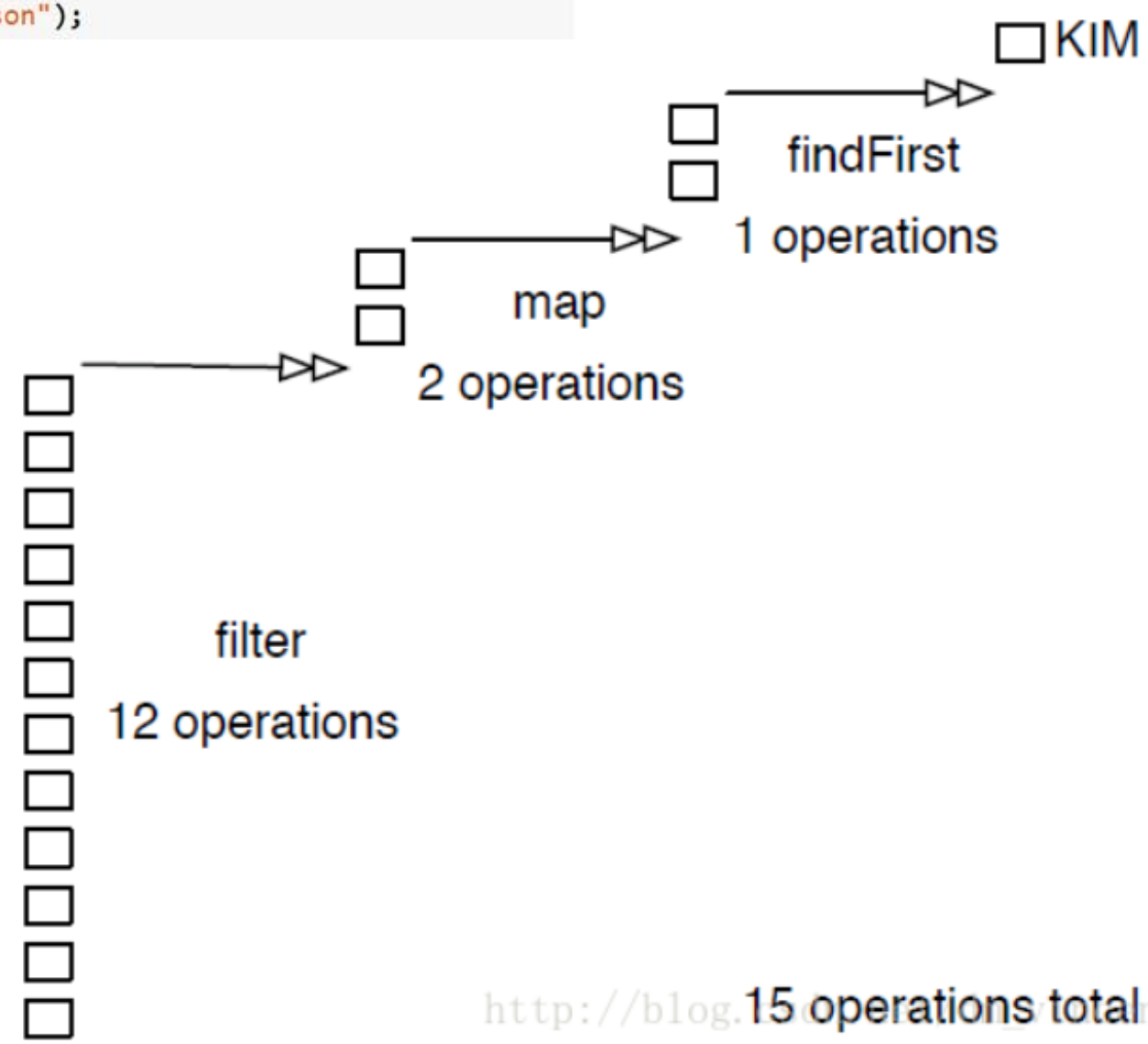
```
        List<String> names = Arrays.asList("Brad", "Kate", "Kim", "Jack", "Joe", "Mike",
"Susan", "George", "Robert", "Julia", "Parker", "Benson");

        final String firstNameWith3Letters = names.stream()
            .filter(name -> length(name) == 3)
            .map(name -> toUpper(name))
            .findFirst()
```

Reference: https://blog.csdn.net/dm_vincent/article/details/40503685

```
List<String> names = Arrays.asList("Brad", "Kate", "Kim", "Jack", "Joe", "Mike",
"Susan", "George", "Robert", "Julia", "Parker", "Benson");
```

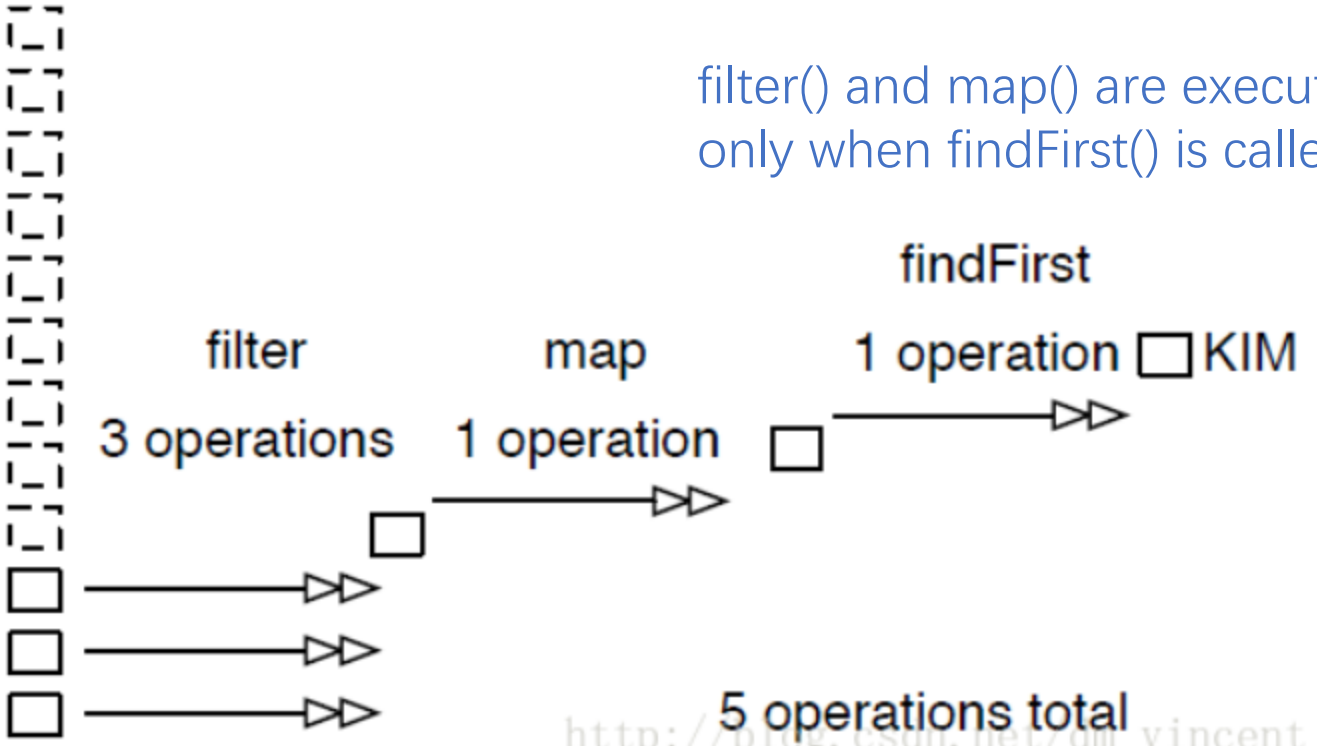
Eager Execution



Reference: https://blog.csdn.net/dm_vincent/article/details/40503685

Lazy Execution

```
List<String> names = Arrays.asList("Brad", "Kate", "Kim", "Jack", "Joe", "Mike",
"Susan", "George", "Robert", "Julia", "Parker", "Benson");
```

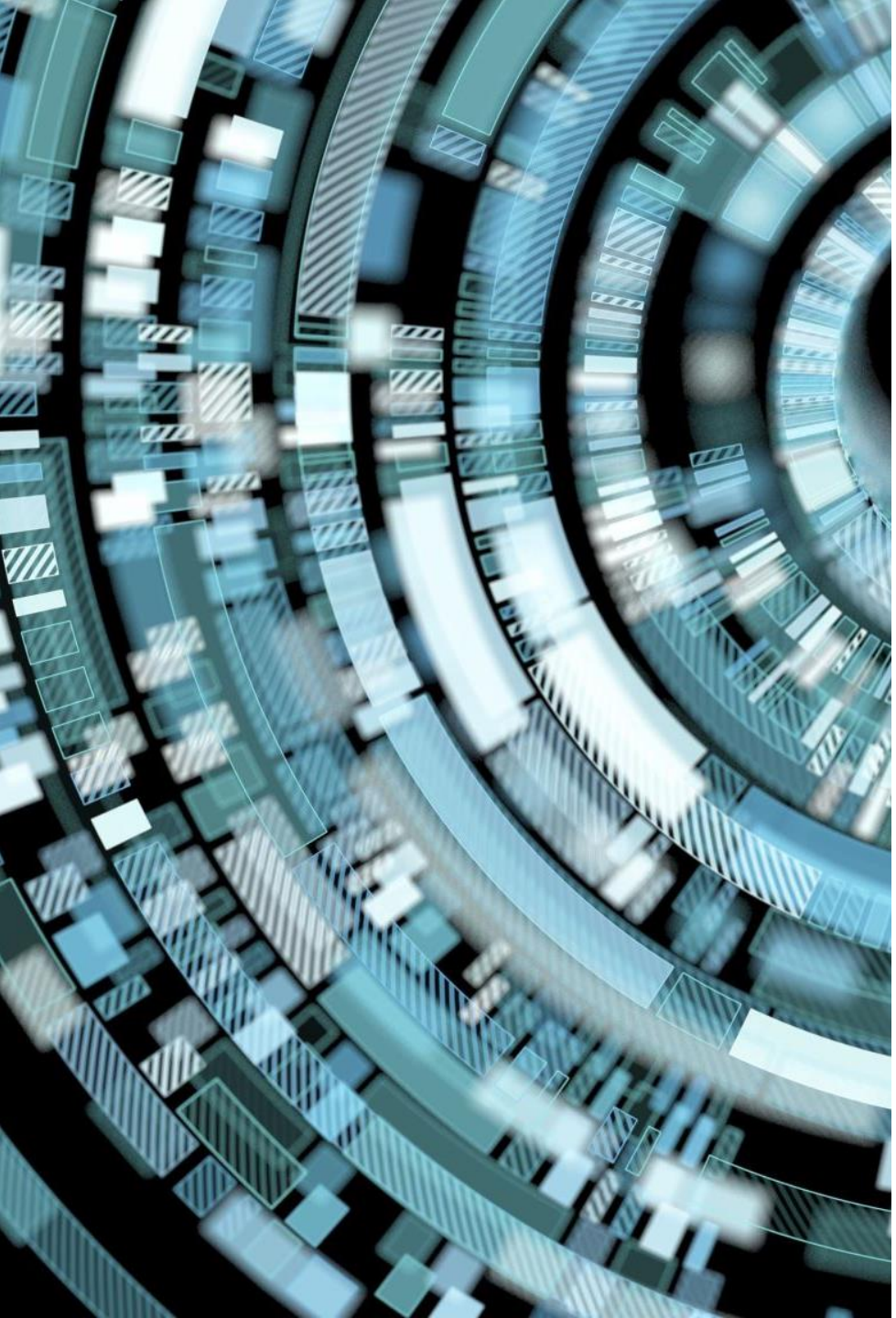


filter() and map() are executed only when findFirst() is called

filter() finds the first matching element and pass it to map()

The calculation terminates as long as we get the result.

Reference: https://blog.csdn.net/dm_vincent/article/details/40503685



Lecture 4

- Streams API
- Optional<T>

Tired of Null Pointer Exceptions? Consider Using Java SE 8's "Optional"!

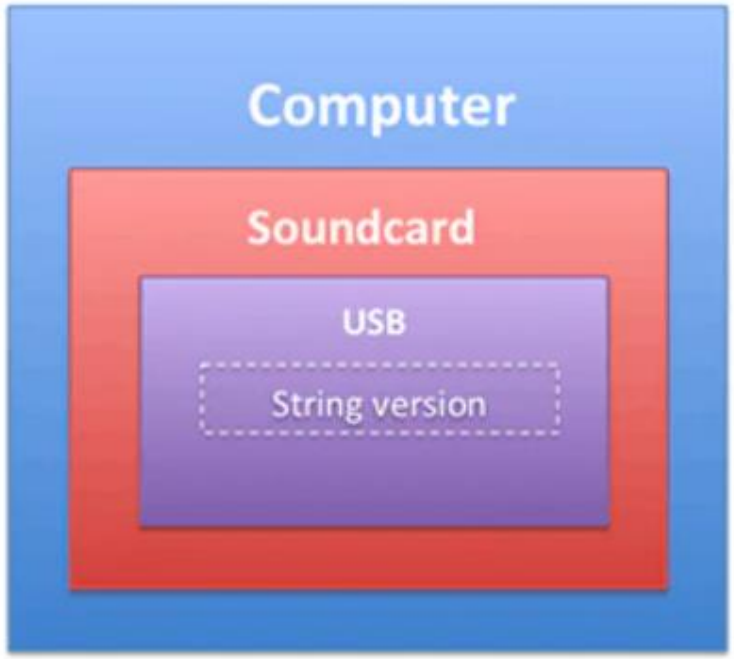
by Raoul-Gabriel Urma
Published March 2014

Make your code more readable and protect it against null pointer exceptions.


```
janv. 10, 2018 10:45:29 AM org.apache.catalina.core.StandardWrapperValve invoke
GRAVE: "Servlet.service()" pour la servlet cs a généré une exception
java.lang.NullPointerException
        at dao.ProduitDaoImpl.ProduitsParMC(ProduitDaoImpl.java:49)
        at web.ControleurServlet.doPost(ControleurServlet.java:47)
        at web.ControleurServlet.doGet(ControleurServlet.java:28)
        at javax.servlet.http.HttpServlet.service(HttpServlet.java:622)
        at javax.servlet.http.HttpServlet.service(HttpServlet.java:729)
        at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:230)
        at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:165)
        at org.apache.tomcat.websocket.server.WsFilter.doFilter(WsFilter.java:52)
        at org.apache.catalina.core.ApplicationFilterChain.internalDoFilter(ApplicationFilterChain.java:192)
        at org.apache.catalina.core.ApplicationFilterChain.doFilter(ApplicationFilterChain.java:165)
        at org.apache.catalina.core.StandardWrapperValve.invoke(StandardWrapperValve.java:198)
        at org.apache.catalina.core.StandardContextValve.invoke(StandardContextValve.java:96)
        at org.apache.catalina.authenticator.AuthenticatorBase.invoke(AuthenticatorBase.java:474)
        at org.apache.catalina.core.StandardHostValve.invoke(StandardHostValve.java:140)
        at org.apache.catalina.valves.ErrorReportValve.invoke(ErrorReportValve.java:79)
        at org.apache.catalina.valves.AbstractAccessLogValve.invoke(AbstractAccessLogValve.java:624)
        at org.apache.catalina.core.StandardEngineValve.invoke(StandardEngineValve.java:87)
        at org.apache.catalina.connector.CoyoteAdapter.service(CoyoteAdapter.java:349)
        at org.apache.coyote.http11.Http11Processor.service(Http11Processor.java:783)
        at org.apache.coyote.AbstractProcessorLight.process(AbstractProcessorLight.java:66)
        at org.apache.coyote.AbstractProtocol$ConnectionHandler.process(AbstractProtocol.java:798)
        at org.apache.tomcat.util.net.NioEndpoint$SocketProcessor.doRun(NioEndpoint.java:1434)
        at org.apache.tomcat.util.net.SocketProcessorBase.run(SocketProcessorBase.java:49)
        at java.util.concurrent.ThreadPoolExecutor.runWorker(Unknown Source)
        at java.util.concurrent.ThreadPoolExecutor$Worker.run(Unknown Source)
        at org.apache.tomcat.util.threads.TaskThread$WrappingRunnable.run(TaskThread.java:61)
        at java.lang.Thread.run(Unknown Source)
```

Prevent Null Pointer Exception (NPE)

Works, but hard to read!




```
String version = "UNKNOWN";
if(computer != null){
  Soundcard soundcard = computer.getSoundcard();
  if(soundcard != null){
    USB usb = soundcard.getUSB();
    if(usb != null){
      version = usb.getVersion();
    }
  }
}
```

```
String version = computer.getSoundcard().getUSB().getVersion();
```

https://www.oracle.com/technical-resources/articles/java/java8-optional.html

The Optional<T> class

- Purpose: a type-level solution for representing optional values instead of `null` references
- A container object which may or may not contain a non-null value (safe alternative for "object or null")
- Help us to specify alternative values to return or alternative actions to take if the value is `null`, without having to use null checkers

```
Optional<String> optionalString = … // the value could be null (e.g., user input)
```

```
String result = optionalString.orElse("");
String result = optionalString.orElseGet(() -> System.getProperty("user.dir"));
String result = optionalString.orElseThrow(IllegalStateException::new);
```

https://horstmann.com/corejava/livelessons2/lesson02/index.html#(23)

Creating Optional Values

```
public final class Optional<T>
extends Object
```

```
static <T> Optional<T> empty()
```

Returns an empty `Optional` instance.

```
static <T> Optional<T> of(T value)
```

Returns an `Optional` with the specified present non-null value.

```
static <T> Optional<T> ofNullable(T value)
```

Returns an `Optional` describing the specified value, if non-null, otherwise returns an empty `Optional`.

Why use Optional.of over Optional.ofNullable?

Asked 7 years, 2 months ago Modified 23 days ago Viewed 186k times

307

When using the Java 8 `Optional` class, there are two ways in which a value can be wrapped in an optional.

```
String foobar = <value or null>;
Optional.of(foobar);         // May throw NullPointerException
Optional.ofNullable(foobar); // Safe from NullPointerException
```

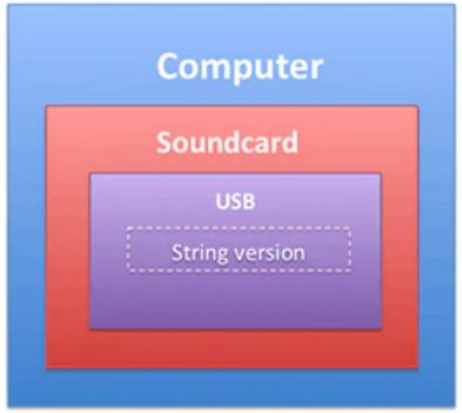
I understand `Optional.ofNullable` is the only safe way of using `Optional`, but why does `Optional.of` exist at all? Why not just use `Optional.ofNullable` and be on the safe side at all times?

415

Your question is based on assumption that the code which may throw `NullPointerException` is worse than the code which may not. This assumption is wrong. If you expect that your `foobar` is never null due to the program logic, it's much better to use `Optional.of(foobar)` as you will see a `NullPointerException` which will indicate that your program has a bug. If you use `Optional.ofNullable(foobar)` and the `foobar` happens to be `null` due to the bug, then your program will silently continue working incorrectly, which may be a bigger disaster. This way an error may occur much later and it would be much harder to understand at which point it went wrong.

https://stackoverflow.com/questions/31696485/why-use-optional-of-over-optional-ofnullable

Example




```java
class Computer {
    private Optional<Soundcard> soundcard;

    public Optional<Soundcard> getSoundcard() {
        return soundcard;
    }
    public void setSoundcard(Optional<Soundcard> soundcard) {
        this.soundcard = soundcard;
    }

}
```

```java
class Soundcard {
    private Optional<USB> usb;

    public void setUsb(Optional<USB> usb) {
        this.usb = usb;
    }
    public Optional<USB> getUsb() {
        return usb;
    }
}

class USB{
    String version;

    public String getVersion() {
        return version;
    }
    public void setVersion(String version) {
        this.version = version;
    }
}
```

Example

```java
public static String getUsbVersion(Computer computer) {
    return computer.getSoundcard() Optional<Soundcard>
            .flatMap(e -> e.getUsb()) Optional<USB>
            .map(e -> e.getVersion()) Optional<String>
            .orElse( other: "UNKNOWN");
}
```

OR

```java
public static String getUsbVersion(Computer computer) {
    return computer.getSoundcard() Optional<Soundcard>
            .flatMap(Soundcard::getUsb) Optional<USB>
            .map(USB::getVersion) Optional<String>
            .orElse( other: "UNKNOWN");
}
```

```java
public static void main(String[] args) {
    USB usb = new USB();
    usb.setVersion("1.0");

    Soundcard soundcard1 = new Soundcard();
    soundcard1.setUsb(Optional.of(usb));
    Soundcard soundcard2 = new Soundcard();
    soundcard2.setUsb(Optional.empty());

    Computer computer1 = new Computer();
    computer1.setSoundcard(Optional.of(soundcard1));

    Computer computer2 = new Computer();
    computer2.setSoundcard(Optional.of(soundcard2));

    Computer computer3 = new Computer();
    computer3.setSoundcard(Optional.empty());

    System.out.println(getUsbVersion(computer1));
    System.out.println(getUsbVersion(computer2));
    System.out.println(getUsbVersion(computer3));
}
```

1.0
UNKNOWN
UNKNOWN

flatMap vs map

```
public <U> Optional<U> map(Function<? super T,? extends U> mapper)
public <U> Optional<U> flatMap(Function<? super T,Optional<U>> mapper)
```

```
public static String getUsbVersion(Computer computer) {
    return computer.getSoundcard()                    Optional<Soundcard>
            .flatMap(Soundcard::getUsb)               Optional<USB>
            .map(USB::getVersion)                     Optional<String>
            .orElse( other: "UNKNOWN");
}
```

```
public static String getUsbVersion(Computer computer) {
    return computer.getSoundcard()                    Optional<Soundcard>
            .map(Soundcard::getUsb)                   Optional<Optional<USB>>
            .map(USB::getVersion)                     Optional<U>
            .orElse( other: "UNKNOWN");
}
```

- Soundcard::getUsb returns `Optional<USB>`
- If using map(), we'll get `Optional<Optional<USB>>`, but we need to invoke USB.getVersion()
- If your function already returns an Optional, use `flatMap()` which doesn't double wrap it

Next Lecture

- I/O Streams
- Character Encoding