

Computer System Design & Application

计算机系统设计与应用A

陶伊达 (TAO Yida)

taoyd@sustech.edu.cn

An abstract graphic on the left side of the slide, featuring concentric circles and various digital patterns like binary code and pixelated shapes in shades of blue, green, and white.

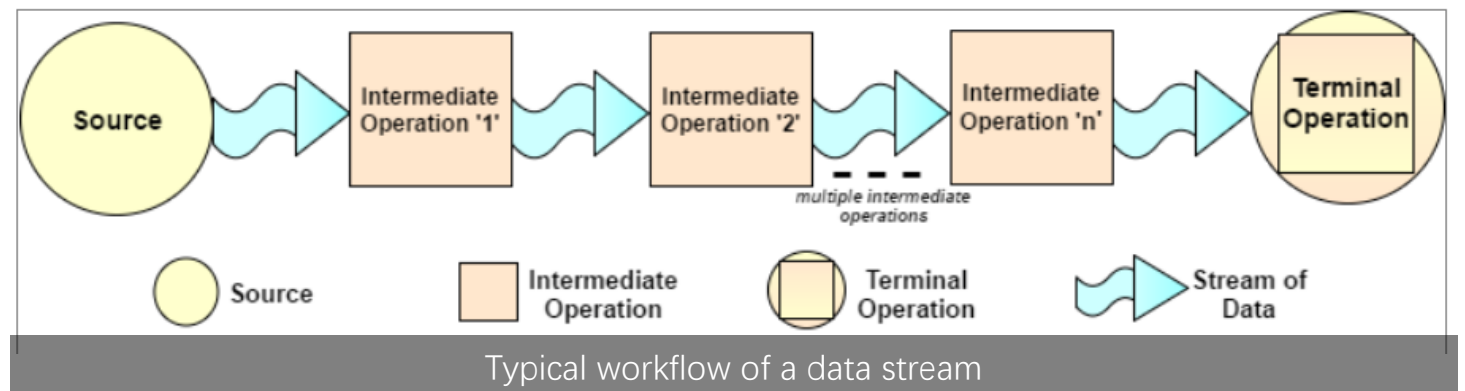
Lecture 4

- Java 8 Stream
- 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**



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> iterate(T seed,  
                             UnaryOperator<T> f)
```

- Approach III: using `Stream.iterate()`, which creates a stream of `seed`, `f(seed)`, `f(f(seed))`, etc.

```
Stream<Integer> evenNumber = Stream.iterate(2, n -> n + 2);
```

```
evenNumber.limit(10).forEach(System.out::println);
```

Create a Stream

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

- Approach IV: using `Stream.of()`, which has a varargs parameter (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");
```


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);
```



Lecture 4

- Java 8 Stream
 - Stream Creation
 - Intermediate Operations
 - Terminal Operations
- Optional<T>

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);
```

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

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

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

Stream Pipeline 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(4,2,3,1,3,5,7,1);  
ilist.stream()  
    .filter(element -> (element % 2 == 1))  
    .map(element -> (element*element))  
    .sorted()  
    .distinct()  
    .limit(3)  
    .forEach(System.out::println);
```




Lecture 4

- Java 8 Stream
 - Stream Creation
 - Intermediate Operations
 - Terminal Operations
- Optional<T>

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 (discuss later)

Reduction

- A reduction is a terminal operation that aggregates a stream into a type or a primitive
- Reduction operations in Java 8 Stream API
 - `min()`
 - `max()`
 - `average()`
 - `sum()`
 - `reduce()`: <- the general one

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

reduce()

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

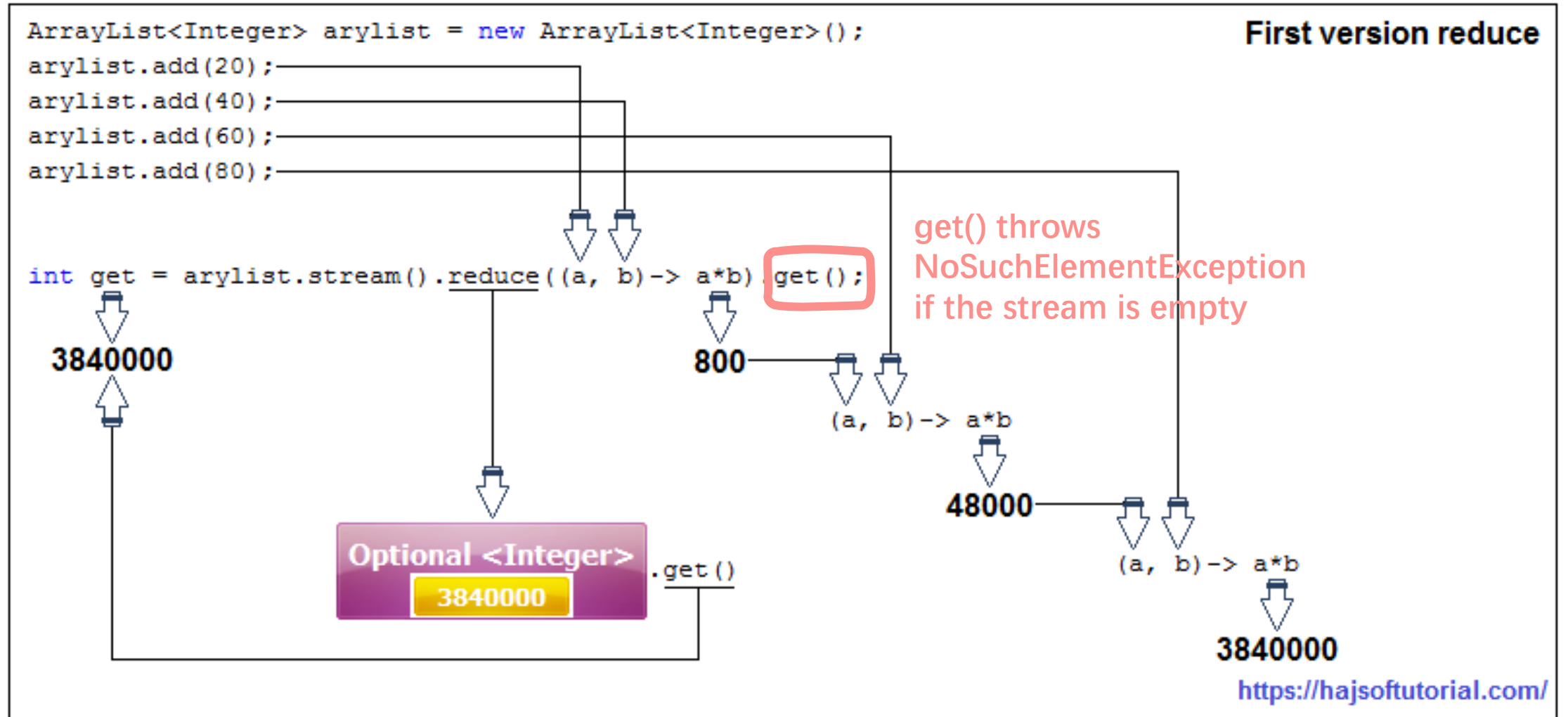
Next element: e.g., the next number in the stream

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

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

Accumulator function: e.g., add two numbers

Example of reduce()



Example of reduce()

- 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

+

•

○

Collecting Results

- When you are done with a stream, you often want to collect the result in a data structure
- `Stream.collect(Collector collector)` performs a **reduction** operation on the elements of this stream using a Collector
- `Collector<T,A,R>` interface
 - **T** - the type of input elements to the reduction operation
 - **R** - the result type of the reduction operation
 - **A** - the mutable accumulation type of the reduction
 - accumulating elements into a Collection
 - concatenating strings
 - computing summary information about elements such as sum, min, max
 -

Collecting Results

- The `Collectors` class provides implementations for various useful reduction operations (i.e., 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`

Grouping & Downstream Collectors

We use `Collectors.groupingBy(Function<T, K> classifier)` returns a collector, which produces `Map<K, List<T>>` that maps elements of type T to some key type K and corresponding elements into a list as the map value

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

```
Map<Integer, List<String>> group = stream.collect(Collectors.groupingBy(String::length));
```

```
{1=[a, a], 2=[bb, cc, bb], 3=[ddd, eee]}
```

Grouping & Downstream Collectors

We could use `Collectors.groupingBy(Function classifier, Collector downstream)` if we want to further perform reduction operations on the list of values

```
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()));
```

```
{1=[a], 2=[bb, cc], 3=[eee, ddd]}
```

Grouping & Downstream Collectors

```
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("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())));
```

Lazy Evaluation

Intermediate operations
are lazily executed

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

Benefits?

Terminal operations are
eagerly executed

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

Example

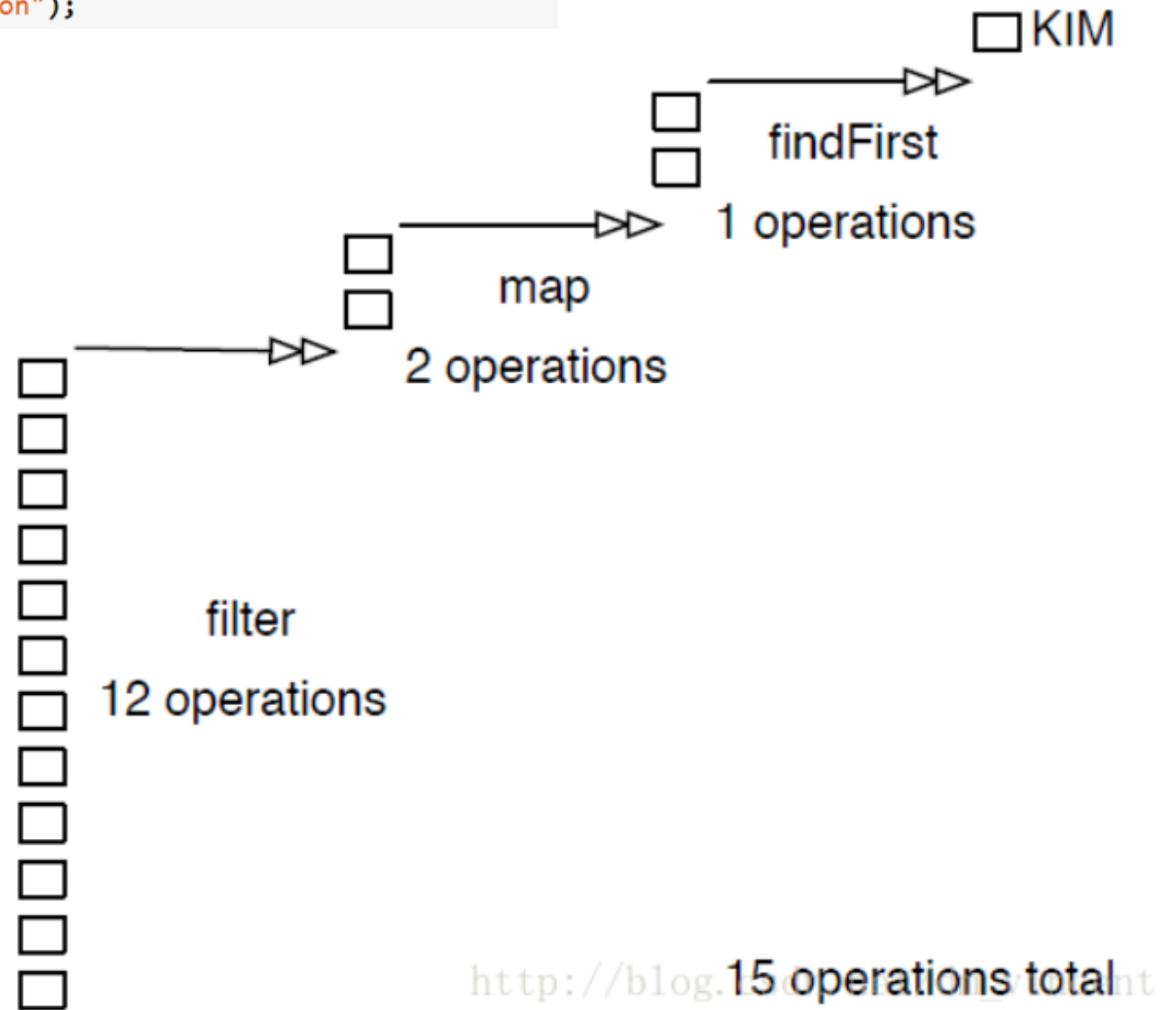
How many operations do we have to perform when the code is executed eagerly / lazily?

```
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

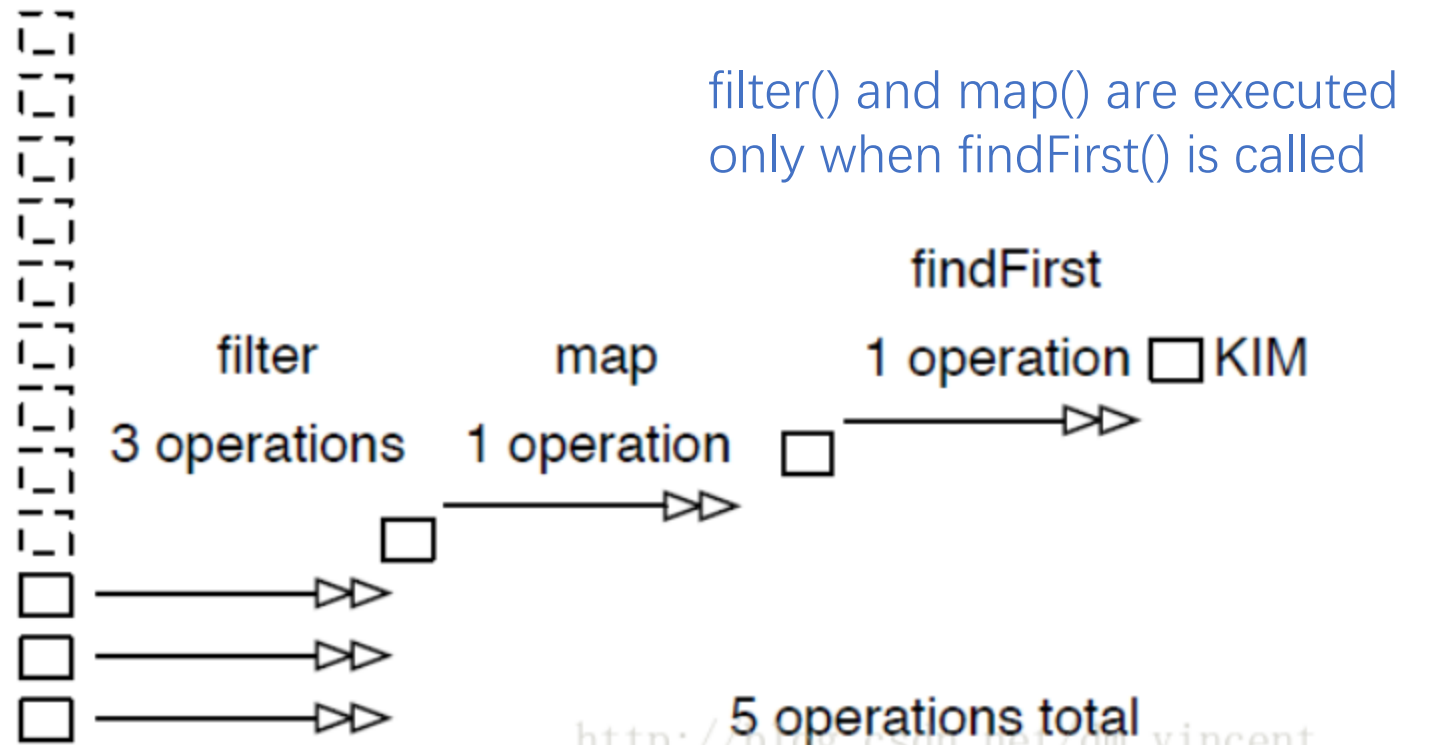
```
List<String> names = Arrays.asList("Brad", "Kate", "Kim", "Jack", "Joe", "Mike",  
"Susan", "George", "Robert", "Julia", "Parker", "Benson");
```

Eager Execution



Lazy Execution

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



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

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

The calculation terminates as long as we get the result.

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"));
```




Lecture 4

- Java 8 Stream
 - Stream Creation
 - Intermediate Operations
 - Terminal Operations
- 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**.

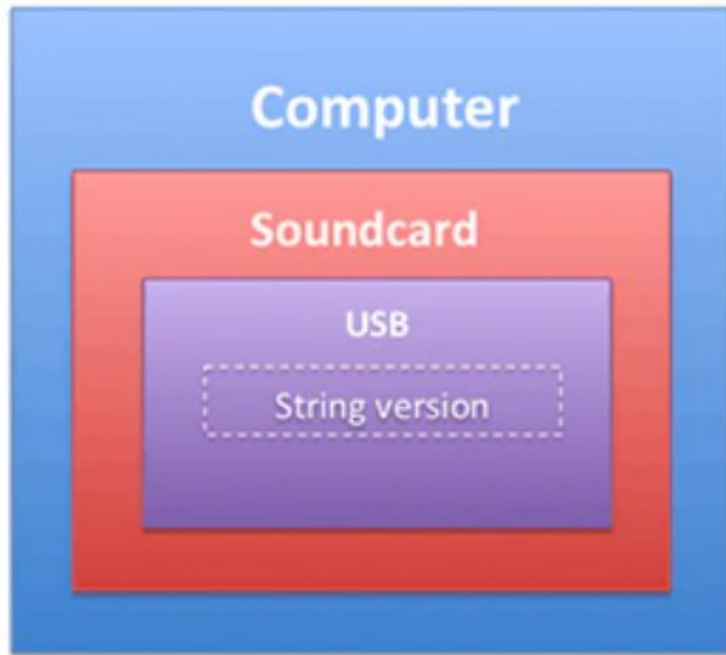
SIMPLY EXPLAINED



NullPointerException

```
janv. 10, 2018 10:45:29 AM org.apache.catalina.core.StandardWrapperValve invoke
GRAVE: "Servlet.service()" pour la servlet cs a g n r  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)



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

Works, but hard to read!

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

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

The Optional<T> class

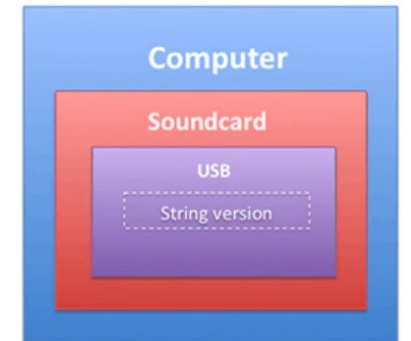
- Purpose: a type-level solution for representing optional values instead of null references
- A container object which may or may not contain a non-null value (safe alternative for “object or null”)
- Help us to specify alternative values to return or **alternative actions** to take if the value is 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#\(23\)](https://horstmann.com/corejava/livelessons2/lesson02/index.html#(23))

Example

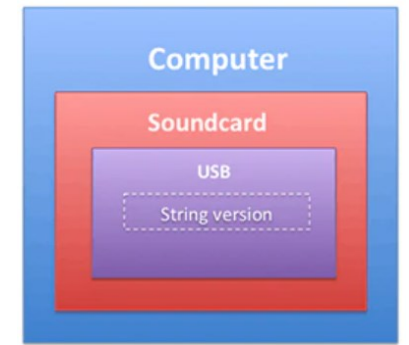


```
class USB{  
    String version;  
  
    public String getVersion() {  
        return version;  
    }  
    public void setVersion(String version) {  
        this.version = version;  
    }  
}
```

```
class Soundcard{  
    Optional<USB> usb;  
  
    public Optional<USB> getUSB(){  
        return usb;  
    }  
    public void setUsb(Optional<USB> usb) {  
        this.usb = usb;  
    }  
}
```


Example

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



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

Why using flatMap instead of map?

Example

```
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.

1.0
UNKNOWN
UNKNOWN

```
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));
```

```
}
```

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");  
}
```

- 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

```
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");  
}
```


Next Lecture

- I/O Streams
- Character Encoding