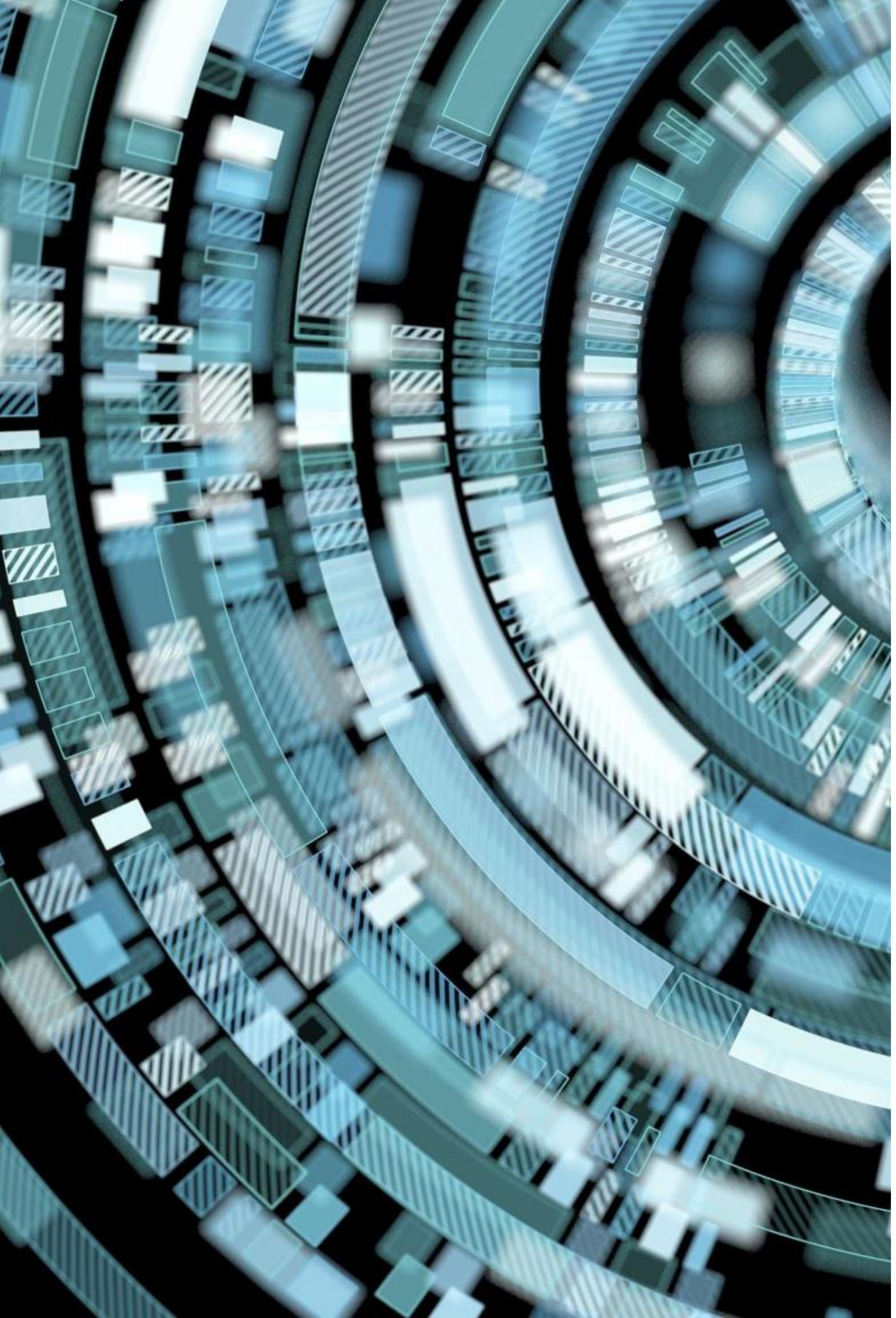
Computer System Design & Application
计算机系统设计与应用A

陶伊达 （TAO Yida）
taoyd@sustech.edu.cn

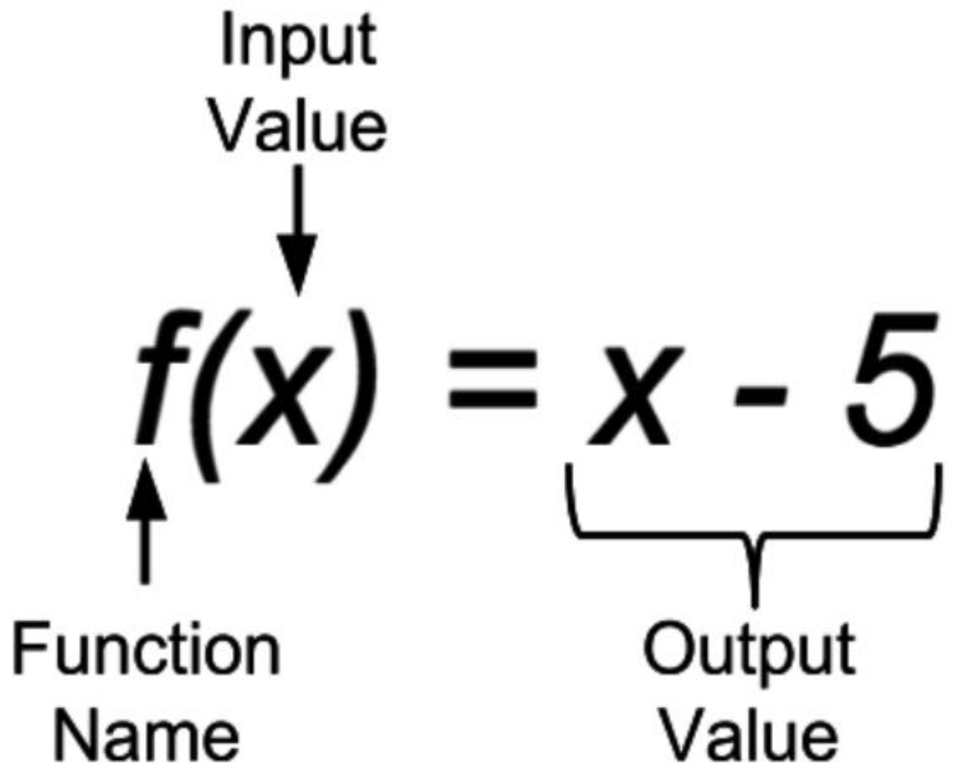


Lecture 4

- Functional Programming
- Lambda Expressions
- Stream API

What is a function?

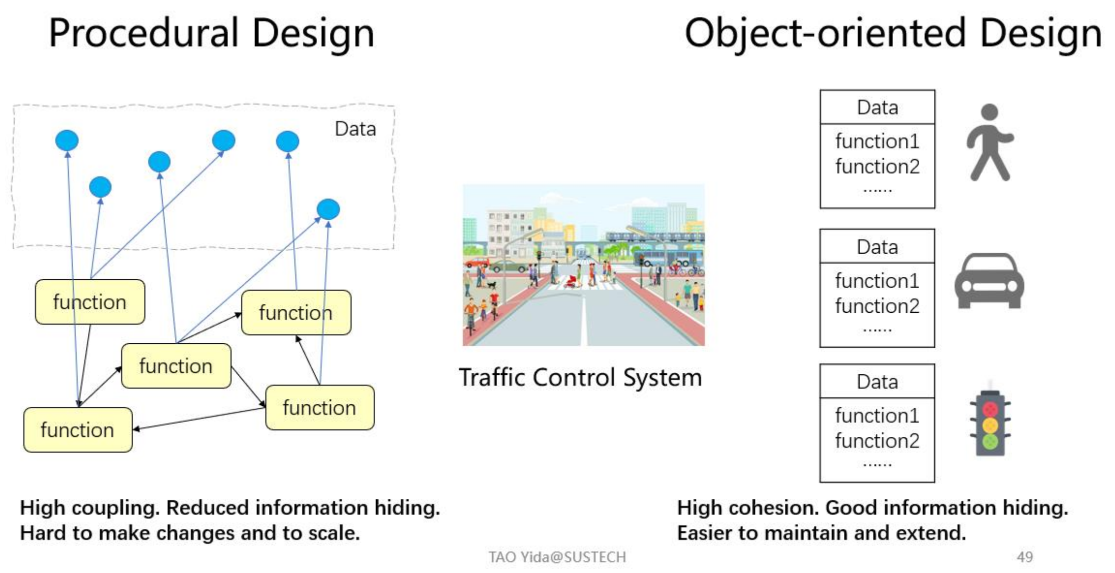
- [Mathematics] A function from a set X to a set Y assigns to each element of X exactly one element of Y
- Maps input to output



x	f(x)
1	-4
2	-3
3	-2
4	-1

What is Functional Programming?

What types of programming paradigms have we seen so far?


Procedural Design

Object-oriented Design

High coupling. Reduced information hiding.
Hard to make changes and to scale.

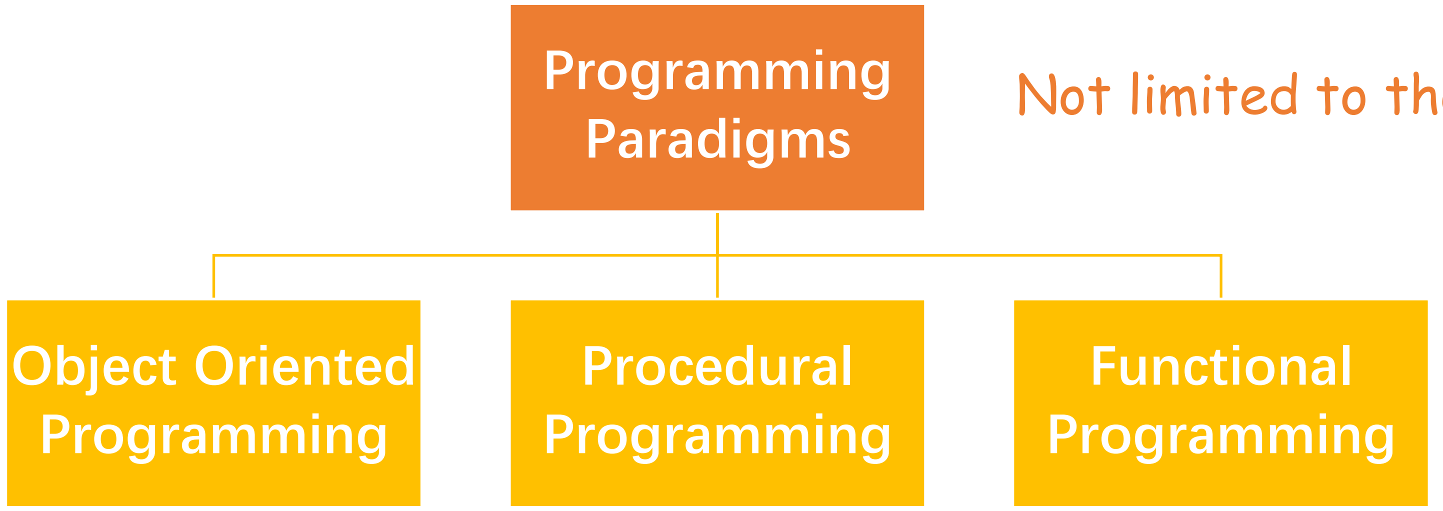
High cohesion. Good information hiding.
Easier to maintain and extend.

TAO Yida@SUSTECH 49

- Functional programming is a **programming paradigm** (编程范式)
- A programming paradigm refers to the way of thinking about things and the way of solving problems

编程范式好比武功门派，博大精深且自成体系。
--摘自《冒号课堂：编程范式与OOP思想》

Programming Paradigms



Not limited to these 3!

Organize data and logics in **objects** (fields and methods)

Organize code as sequential **procedures** (may change same data)

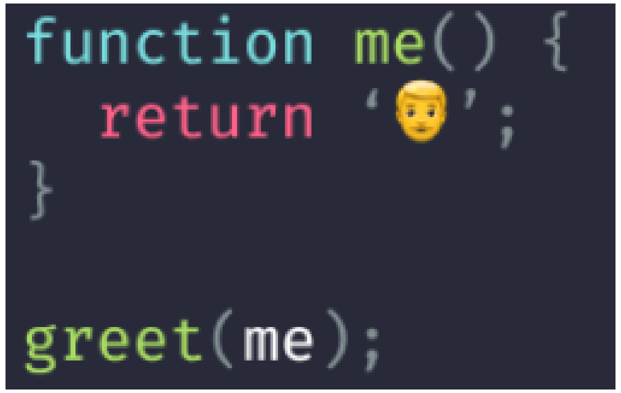
Organize code as **functions** (behaviors are separated from the data, which won't be changed)

Functional Programming (函数式编程)

- Basic unit of computation is function
- Primary characteristics
  - Functions are treated as first-class citizens
  - No side effects
  - Referential transparency
  - Immutability
  - Recursion

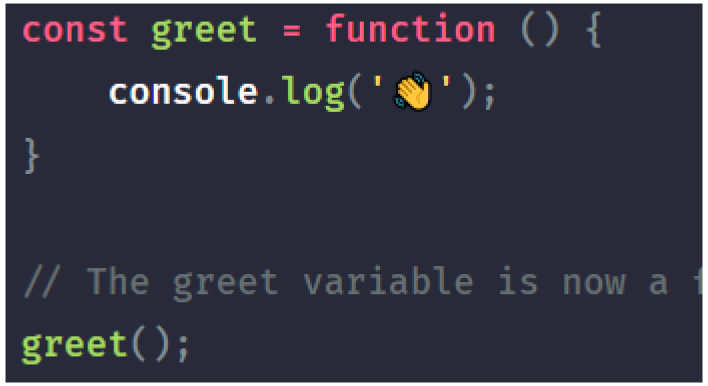
First-class functions

Functions are treated like any other variables



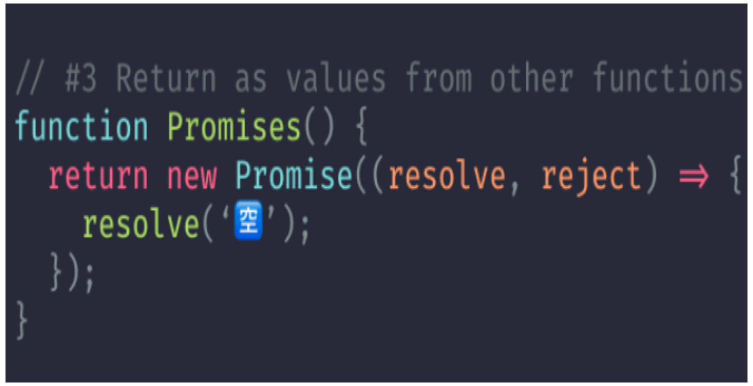
```
function me() {
  return '👱';
}

greet(me);
```

Functions can be passed as arguments



```
const greet = function () {
    console.log('👋');
}

// The greet variable is now a
greet();
```

Functions can be assigned to a variable



```
// #3 Return as values from other functions
function Promises() {
  return new Promise((resolve, reject) => {
    resolve('🈳');
  });
}
```

Functions can be returned

Image source : https://www.webtips.dev/webtips/javascript-interview/first-class-functions

Higher Order Functions

- A higher order function
  - Takes another function as argument
  - Returns another function as result

- A common usage scenario: data processing
  - `map()`, `filtering()`, `reduce()`, etc.
  - More on these later

No Side Effects

- Side effects: Events that are caused by a system with a **limited scope**, whose effects are felt **outside of that scope**
- Pure functions have no side effects (cannot change external states)

Impure function	**Side Effects**
writeFile(filename)	External files are changed
updateDatabase(table)	External database table is changed
sendAjaxRequest(request)	External server state is changed

No Side Effects

- Pure functions **always** produce the same output for the same input (regardless of the history)

Impure function	Input	Possible Output
writeFile(filename)	filename	Success or failures given file state
queryDatabase(table)	table	Different results given table state
sendAjaxRequest(request)	request	Different responses given server state

No Side Effects

- Pure functions **always** produce the same output for the same input (regardless of the history)

Is this a pure function?

```
global_list = []
def append_to_list(x):
    global_list.append(x)
    print(global_list)

append_to_list(1)
global_list: [1]

append_to_list(1)
global_list: [1,1]
```

Side effects

- `global_list` is implicitly changed even though it is not declared as the input to the function

- The output of the function changed even though the input remains the same

Referential Transparency

- Functional programs do not have assignment statements
- Variables, once defined, never change their values (eliminate side effects)

```
x = x + 10
```

Assignment is NOT referentially transparent

```
int plusTen(int x)
{
        return x+10;
}
```

Referentially transparent

Recursion

- No “while” or “for” loop in functional programming
- How do we perform iterations though?

$$n! = n \cdot (n-1) \cdot (n-2) \cdots 3 \cdot 2 \cdot 1$$

```
def factorial(n):
    fact = 1
    while n >= 1:
        fact = fact * n
        n = n - 1
    return fact
```

Using loops

```
def factorial(n):
    if n <= 0: return 1
    return n * factorial(n-1)
```

Using recursive functions (which invoke themselves)

Advantages of Functional Programming

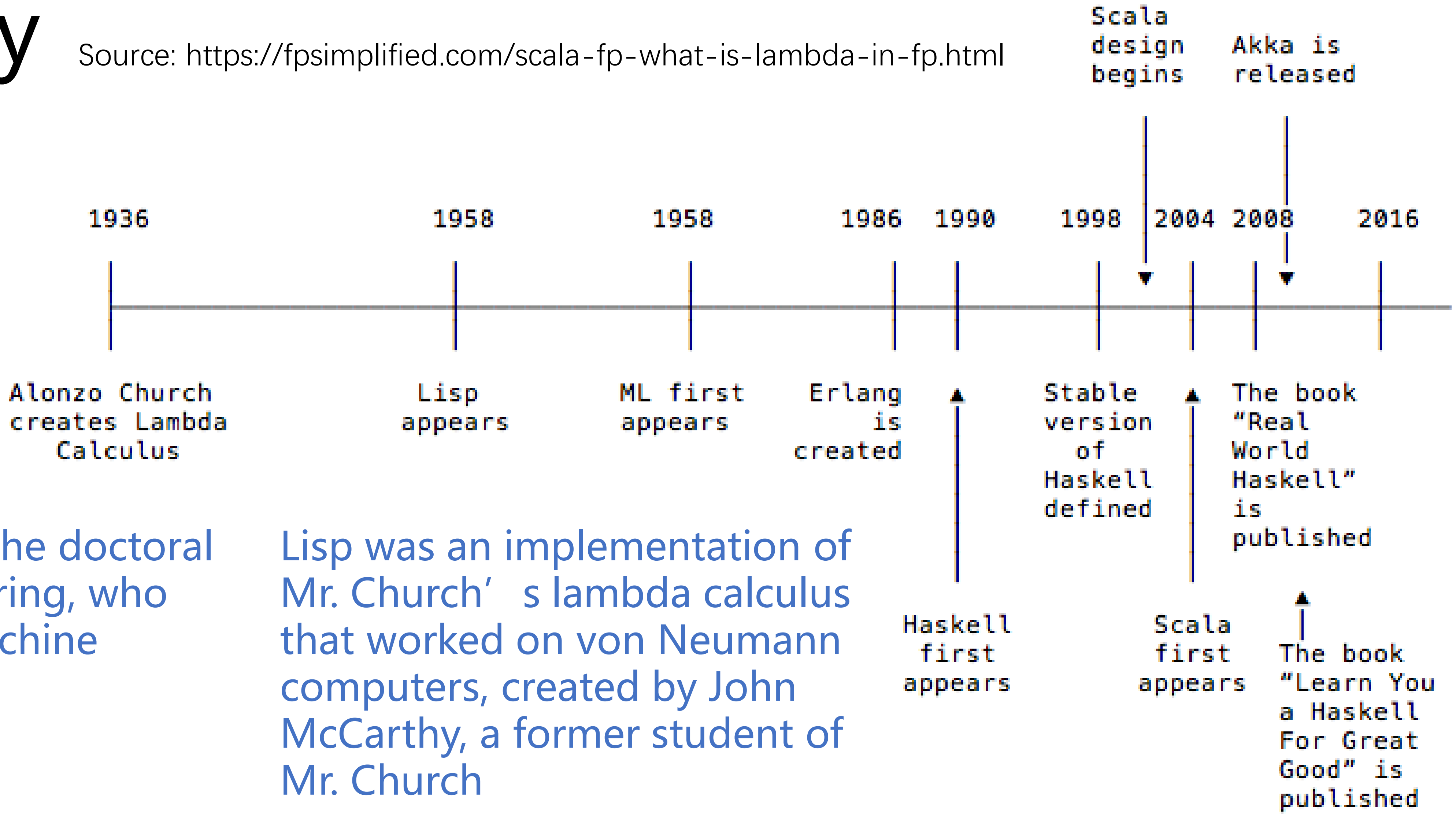
- Easy to debug, test, and parallelize
  - Same input => same output (deterministic)
  - No side effects
  - Data are immutable
- Complexity is dramatically reduced (architectural simplicity)
  - The only interaction with the external system is via the argument and return value (API)

OO style: Object methods interact with the object states

Procedural style: external state is often manipulated from within the function

History

Source: https://fpsimplified.com/scala-fp-what-is-lambda-in-fp.html


Scala design begins
Akka is released
1936
1958
1958
1986
1990
1998
2004
2008
2016
Alonzo Church creates Lambda Calculus
Lisp appears
ML first appears
Erlang is created
Stable version of Haskell defined
The book "Real World Haskell" is published
Haskell first appears
Scala first appears
The book "Learn You a Haskell For Great Good" is published

Alonzo Church is the doctoral advisor of Alan Turing, who created Turing Machine

Lisp was an implementation of Mr. Church' s lambda calculus that worked on von Neumann computers, created by John McCarthy, a former student of Mr. Church

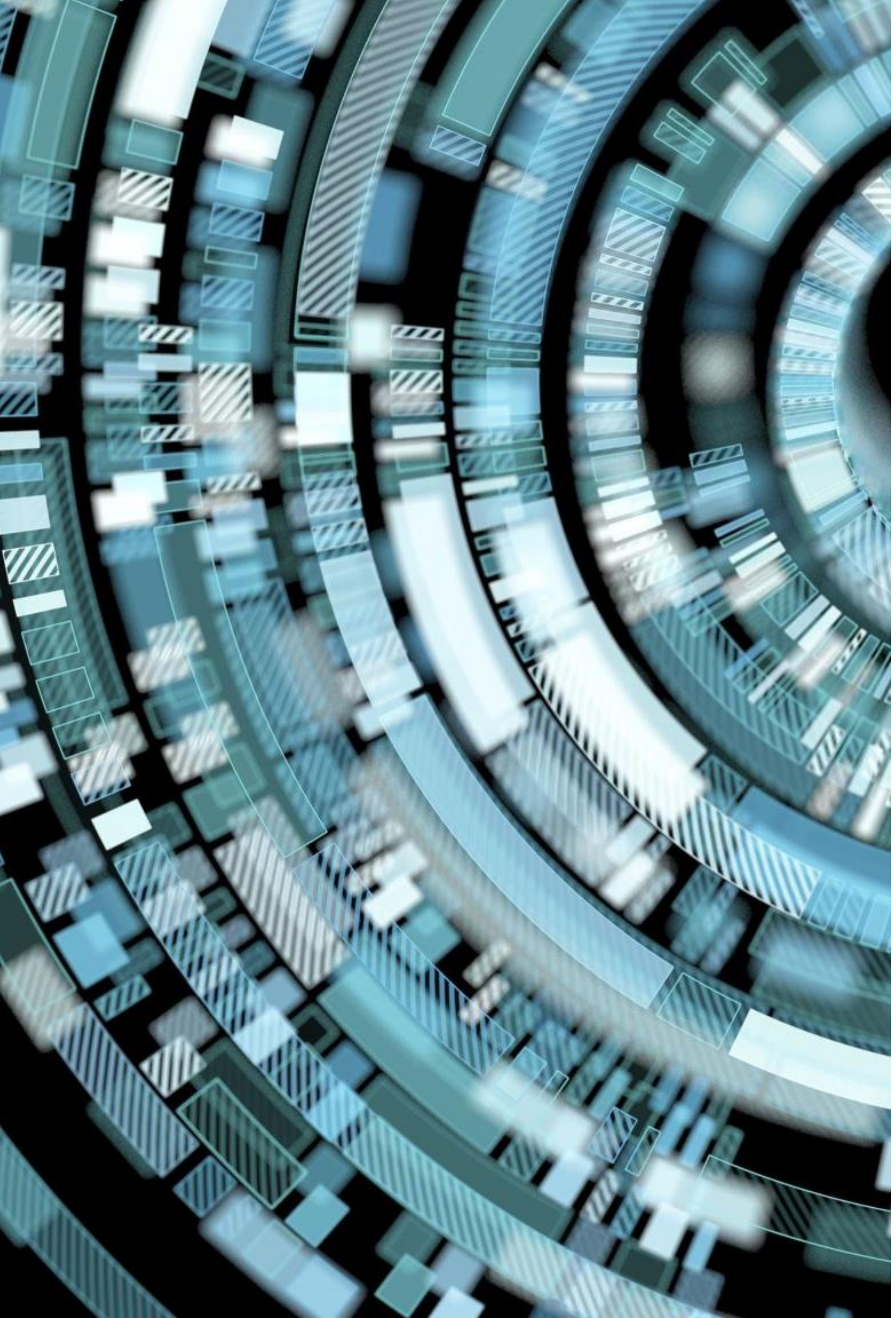
Functional Programming Languages

- Lisp, Erlang, Haskell, Clojure, Scala, F#, Python, Javascript, Kotlin, Rust, Swift, etc.

- FP is used in big companies
    - **Whatsapp** needs only 50 engineers for its 900M users, using Erlang
    - **Huawei** adopts Rust to engineer trustworthy software systems
    - **NVIDIA** uses Haskell for the backend development of its GPUs
    - **Facebook** uses Haskell to fight spams
    - ……

Functional Programming in Java

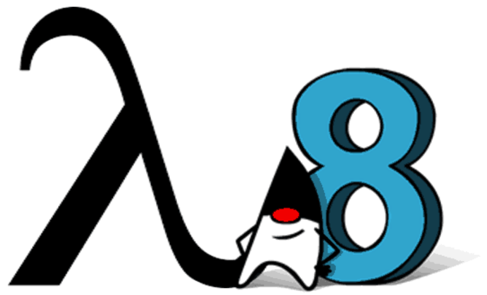
- Different programming paradigms are not necessarily mutually exclusive
    - Python also supports OOP
    - Java also supports the functional styles of programming
- Java 8 introduces functional programming abilities
    - Lambda expressions
    - Streams API



Lecture 4

- Functional Programming
- Lambda Expressions
  - Syntax
  - Use cases
  - Type inference
  - Method references
  - Java Functional Interfaces
- Stream API

Java Lambda Expressions



- Introduced in Java 8
  - Java's first step into functional programming
- A Java lambda expression
  - is an anonymous function with no name/identifier
  - can be created without belonging to any class
  - can be passed as a parameter to another function
  - are callable anywhere in the program

Lambda Syntax

Arrow token

```
(param1, param2) -> {lambda expressions}
```

Left part – lambda parameters

- No function name
- Parentheses could be omitted for a single parameter
- Multiple parameters are separated by comma (,)
- `()` is used if no parameter is needed

Right part – lambda expressions

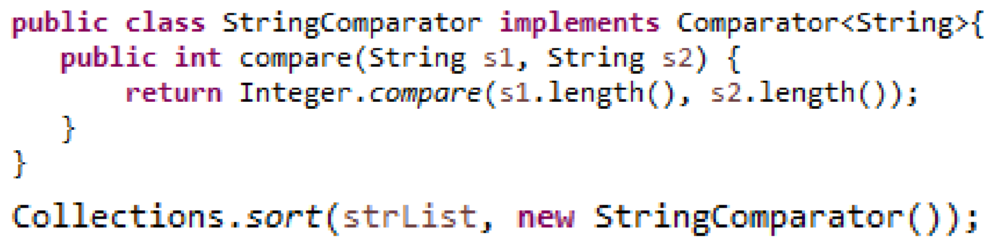
- Curly braces could be omitted for a single expression
- Multiple expressions are separated by a semi-colon (;)
- Can have a `return` statement
- Cannot contain variables, assignments, statements such as `if` or `for`

Use Cases

- Replacing functional interfaces (discussed here)
- Used with the `Streams API` for data processing (next section)

- Functional interface
  - An interface with a single abstract method
  - Before Java 8, functional interfaces are often represented using the anonymous class

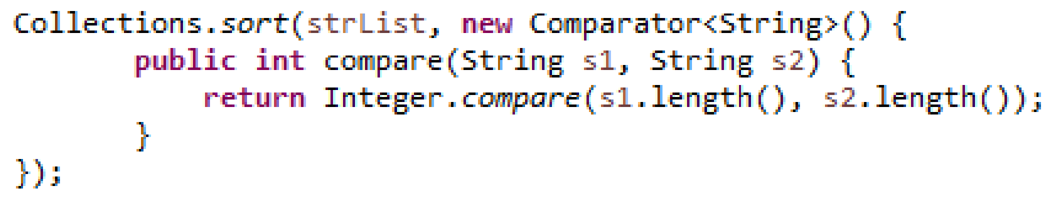
Task: sorting a string list by element's length



```
public class StringComparator implements Comparator<String>{
   public int compare(String s1, String s2) {
       return Integer.compare(s1.length(), s2.length());
   }
}
Collections.sort(strList, new StringComparator());
```

Classic way

- Explicitly creating a class that implements the Comparator interface
- Verbose



```
Collections.sort(strList, new Comparator<String>() {
        public int compare(String s1, String s2) {
            return Integer.compare(s1.length(), s2.length());
        }
});
```

Using the anonymous class

- Don't have to declare a name for it
- Declare and instantiate the class at the same time
- Anonymous class can be used only once
- Shorter code, but still verbose

Using lambda in Java 8

```
Collections.sort(strList, (s1, s2) -> Integer.compare(s1.length(), s2.length()));
```

Matching Lambdas to Functional Interfaces

```
Collections.sort(strList, (s1, s2) -> Integer.compare(s1.length(), s2.length()));
```

```
Collections.sort(List<T> list,          Comparator<? super T> c )
```

1. Lambda is matched to the Comparator interface

```
int compare(T o1, T o2)
```

2. Lambda is matched to `Comparator.compare(T o1, T o2)` method, which is the only abstract(unimplemented) method in the `Comparator` interface

3. The parameter and return type are deduced by compiler using type inference

Matching Lambdas to Functional Interfaces

1. Matching lambda to interface
    - By comparing caller method’s parameter type

2. Matching lambda to interface’s abstract method
    - The interface must have only one abstract (unimplemented) method

3. Matching method parameters and return type
    - Parameter types and return type must match

All of these must be satisfied to replace a functional interface with lambda successfully!

Matching lambda to interface's abstract method

- The interface must have only one abstract (unimplemented) method
  - But it can have multiple default and static methods
- From Java 8, an interface could have both default methods and static methods (method with implementations)

```
public interface MyInterface {

    void printIt(String text);

    default public void printUtf8To(String text, OutputStream outputStream){
        try {
            outputStream.write(text.getBytes("UTF-8"));
        } catch (IOException e) {
            throw new RuntimeException("Error writing String as UTF-8 to OutputStream", e);
        }
    }

    static void printItToSystemOut(String text){
        System.out.println(text);
    }
}
```

Source: http://tutorials.jenkov.com/java/lambda-expressions.html

Example

- **This interface can be implemented by lambda**
- **Although it has 3 methods, only 1 is unimplemented**

Lambda Parameters

- Zero Parameter

```
() -> System.out.println("No parameter");
```

- One Parameter

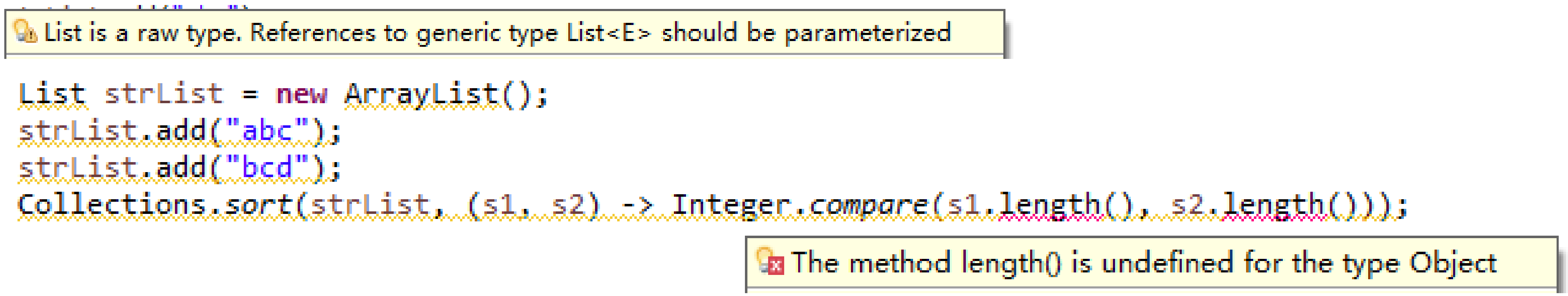
```
(param) -> System.out.println("1 parameter:"+ param);
param -> System.out.println("1 parameter:"+ param);
```

- Multiple Parameters

```
(param1, param2) -> System.out.println("2 parameters:" + param1 + ", " + param2);
```

Type Inference

- Compiler obtains most of the type information from *generics*
- Compiler won't be able to infer types if raw type (e.g., `List`) is used instead of the parameterized type (e.g., `List<String>`)



List is a raw type. References to generic type List<E> should be parameterized

```
List strList = new ArrayList();
strList.add("abc");
strList.add("bcd");
Collections.sort(strList, (s1, s2) -> Integer.compare(s1.length(), s2.length()));
```

The method length() is undefined for the type Object

Use `List<String> strList = new ArrayList<String>()` to remove all the warnings and errors!

Lambda Function Body

- One statement

```
(param) -> System.out.println("1 parameter:"+ param);
```

- Multiple statements (with curly braces)

```
(param) -> {
     System.out.println("1 parameter:"+ param);
     return 0;
}
```

- Return

```
(param1, param2) -> {return param1 > param2;}
(param1, param2) -> param1 > param2;
```

Does the code work?

```java
String s1 = "";
Comparator<String> comp = (s1, s2) -> s1.length() - s2.length();
```

Lambda expression's parameter s1 cannot redeclare another local variable defined in an enclosing scope.

Local variable str defined in an enclosing scope must be final or effectively final

```java
String str = "";
Comparator<String> comp = (s1, s2) -> {  str = str + " test";
                                         return s1.length() - s2.length();};
```

```java
String str = "";
Comparator<String> comp = (s1, s2) -> { System.out.println(str);
                                         return s1.length() - s2.length();};
```

√ The local variable could be accessed without changing its value

More Use Cases

- Use Case I: create & “instantiate” a functional interface

```
public interface MyInterface{

    // abstract method
    double getPiValue();

}


MyInterface ref = () -> 3.1415;
System.out.println("Pi = " + ref.getPiValue());}
```

- The lambda expression implements the abstract method; therefore, we could “instantiate” the interface
- Strictly, we are instantiating an anonymous class that implements the interface

More Use Cases

- Use Case II: executing the same operation when iterating elements

```java
List<String> strList = new ArrayList<String>();
strList.add("abc");
strList.add("bcd");

//print every element in the list
strList.forEach(elem -> System.out.println(elem));
```

```java
public void forEach(Consumer<? super E> action)
```

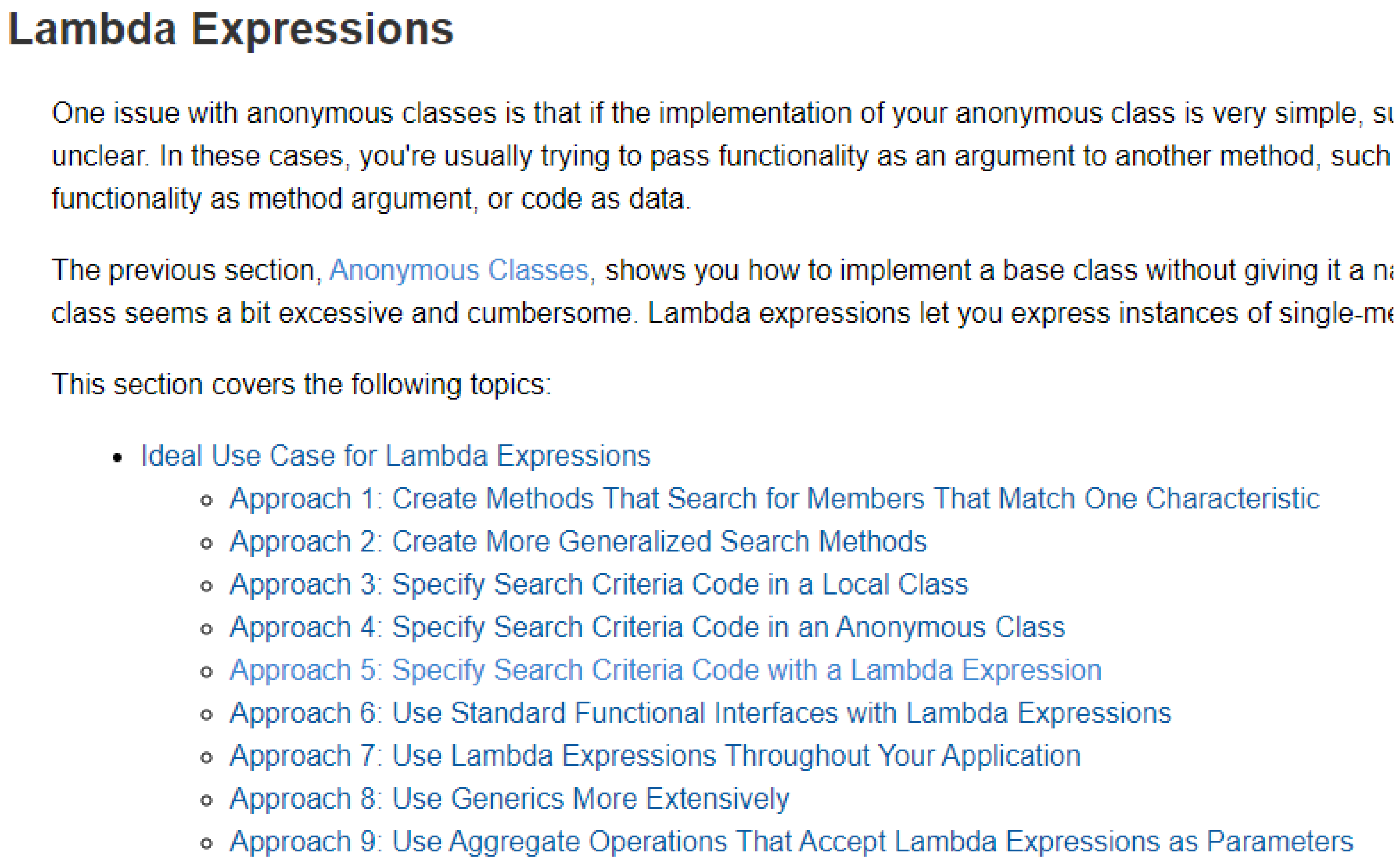
Only 1 abstract method

```java
void accept(T t)
```

Performs this operation on the given argument.

More Use Cases

https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html#approach5



Lambda Expressions

One issue with anonymous classes is that if the implementation of your anonymous class is very simple, su
unclear. In these cases, you're usually trying to pass functionality as an argument to another method, such
functionality as method argument, or code as data.

The previous section, Anonymous Classes, shows you how to implement a base class without giving it a na
class seems a bit excessive and cumbersome. Lambda expressions let you express instances of single-me

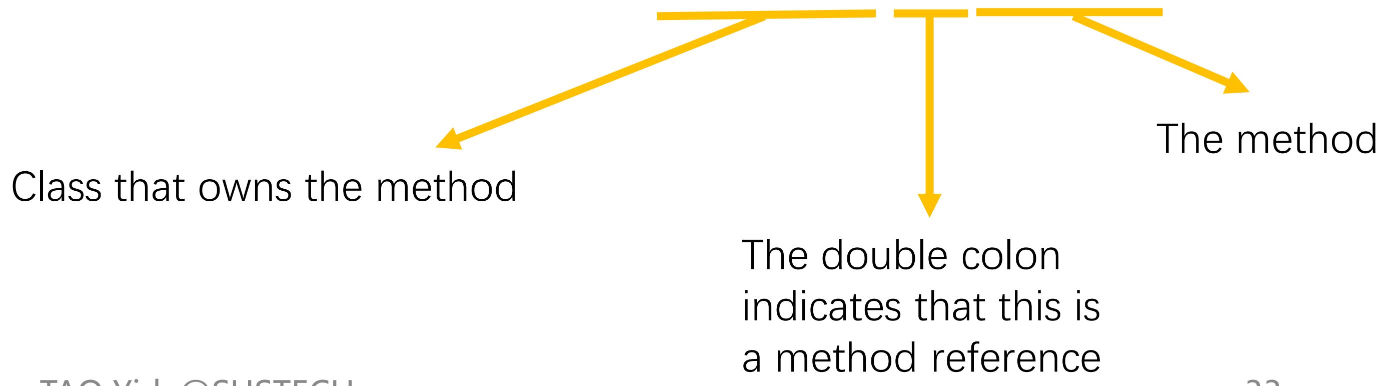
This section covers the following topics:

- Ideal Use Case for Lambda Expressions
  - Approach 1: Create Methods That Search for Members That Match One Characteristic
  - Approach 2: Create More Generalized Search Methods
  - Approach 3: Specify Search Criteria Code in a Local Class
  - Approach 4: Specify Search Criteria Code in an Anonymous Class
  - Approach 5: Specify Search Criteria Code with a Lambda Expression
  - Approach 6: Use Standard Functional Interfaces with Lambda Expressions
  - Approach 7: Use Lambda Expressions Throughout Your Application
  - Approach 8: Use Generics More Extensively
  - Approach 9: Use Aggregate Operations That Accept Lambda Expressions as Parameters

Method Reference

- Sometimes, a lambda expression does nothing but call an existing method
- Method reference allows us to refer to this existing method by name, which is often (but not always) easier to read

```
public interface MyInterface{
        public void print(String s);
}


// Using lambda expression
MyInterface ref = s -> System.out.println(s);

// Using method reference
MyInterface ref = System.out::println;
```


Class that owns the method
The double colon indicates that this is a method reference
The method

What types of methods can be referenced?

- Static method
- Instance method (Bound)
- Instance method (Unbound)
- Constructor

What types of methods can be referenced?

- Static method
- Instance method (Bound)
- Instance method (Unbound)
- Constructor

Example

```
Integer::parseInt
```

Lambda Equivalent

```
str -> Integer.parseInt(str)
```

What types of methods can be referenced?

- Static method
- Instance method (Bound)
- Instance method (Unbound)
- Constructor

```
public interface Deserializer {
    public int deserialize(String v1);
}
```

```
public class StringConverter {
    public int convertToInt(String v1){
        return Integer.valueOf(v1);
    }
}
```

```
StringConverter stringConverter = new StringConverter();

Deserializer des = stringConverter::convertToInt;
```

- `convertToInt` and `deserialize` has the same signature
- Create a `StringConverter` instance and refer to its `convertToInt` method

Example from http://tutorials.jenkov.com/java/lambda-expressions.html

What types of methods can be referenced?

- Static method
- Instance method (Bound)
- **Instance method (Unbound)**
- Constructor

Example

```
String::toLowerCase
String::indexOf
```

Lambda Equivalent

```
s -> s.toLowerCase()
(s1,s2) -> s1.indexOf(s2)
```

The **object** of the referenced method is supplied as **the first of** the lambda's parameters

What types of methods can be referenced?

- Static method
- Instance method (Bound)
- Instance method (Unbound)
- Constructor

Example

```
Supplier<String> s  = String::new
```

Lambda Equivalent

```
Supplier<String> s  = () ->  new String();
```

Same signature

```
public interface Supplier<T>
```

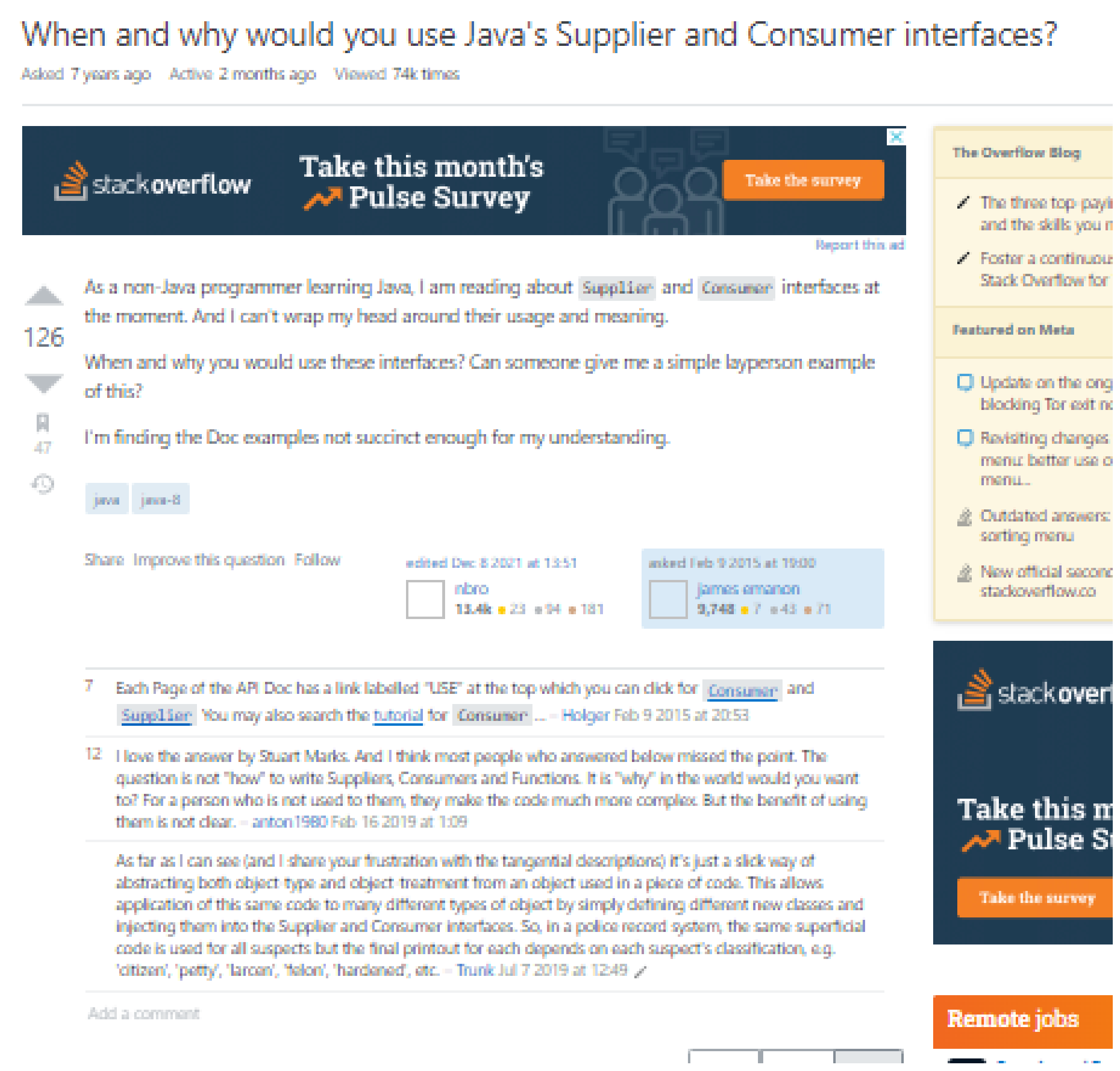
The `Supplier` interface has one abstract method `T get()`

Built-in Functional Interfaces

- The `java.util.function` package
- Well defined set of general-purpose functional interfaces
  - All have only one abstract method
  - Lambda can be used wherever these interfaces are used
  - They are used extensively in Java class libraries, especially with the Streams API (later)
- We've already met them
  - `Consumer<T>` (page 31)
  - `Supplier<T>` (page 38)

Why do we need functional interfaces?

- Think of it as simply “functions”
- Functions can be used as parameters, and simplified using lambdas
- A functional interface defines the structure of that function (i.e., parameter types and return type)



When and why would you use Java's Supplier and Consumer interfaces?

Asked 7 years ago Active 2 months ago Viewed 74k times

126

As a non-Java programmer learning Java, I am reading about `Supplier` and `Consumer` interfaces at the moment. And I can't wrap my head around their usage and meaning.

When and why you would use these interfaces? Can someone give me a simple layperson example of this?

I'm finding the Doc examples not succinct enough for my understanding.

java java-8

Share Improve this question Follow — edited Dec 8 2021 at 13:51 nbro 13.4k — asked Feb 9 2015 at 19:00 james emanon 9,748

7 Each Page of the API Doc has a link labelled "USE" at the top which you can click for `Consumer` and `Supplier` You may also search the tutorial for `Consumer` ... – Holger Feb 9 2015 at 20:53

12 I love the answer by Stuart Marks. And I think most people who answered below missed the point. The question is not "how" to write Suppliers, Consumers and Functions. It is "why" in the world would you want to? For a person who is not used to them, they make the code much more complex. But the benefit of using them is not clear. – anton1980 Feb 16 2019 at 1:09

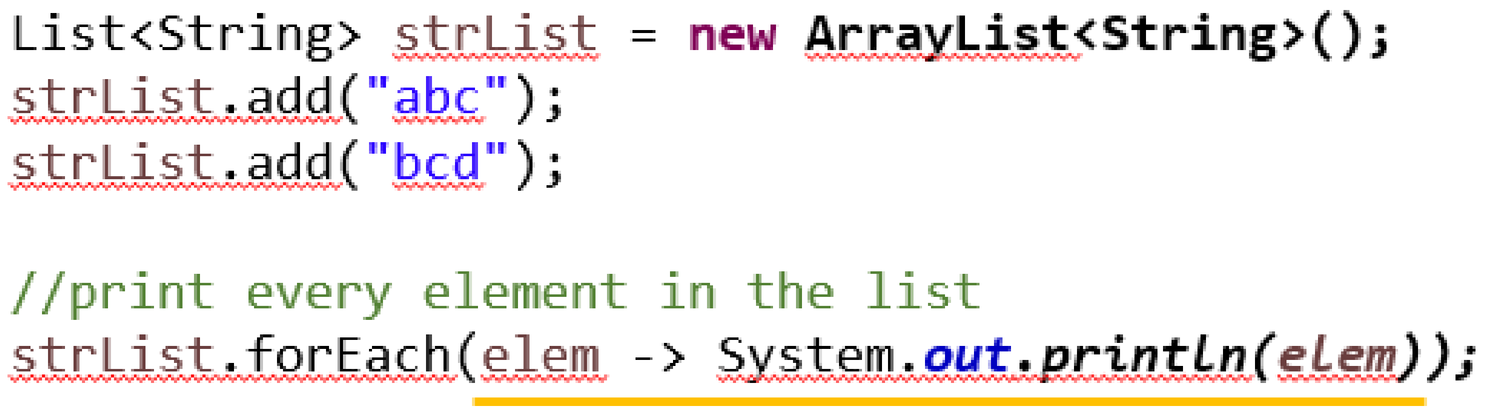
As far as I can see (and I share your frustration with the tangential descriptions) it's just a slick way of abstracting both object type and object treatment from an object used in a piece of code. This allows application of this same code to many different types of object by simply defining different new classes and injecting them into the Supplier and Consumer interfaces. So, in a police record system, the same superficial code is used for all suspects but the final printout for each depends on each suspect's classification, e.g. 'citizen', 'petty', 'larcen', 'felon', 'hardened', etc. – Trunk Jul 7 2019 at 12:49

Add a comment

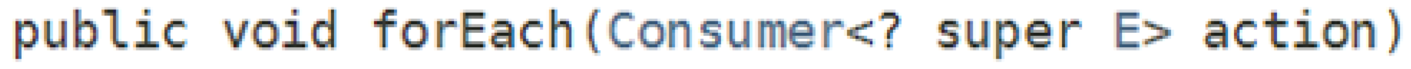
https://stackoverflow.com/questions/28417262

Consumer<T>

represents a function that takes an argument and returns nothing (consume it)



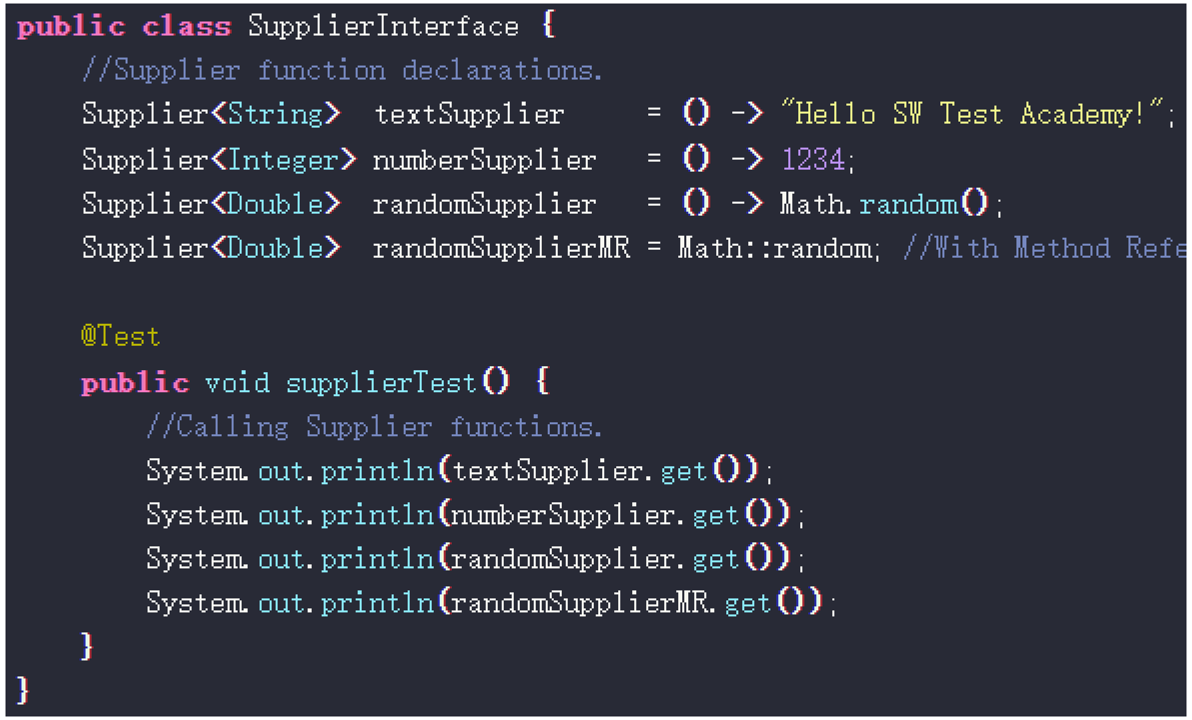
```java
List<String> strList = new ArrayList<String>();
strList.add("abc");
strList.add("bcd");

//print every element in the list
strList.forEach(elem -> System.out.println(elem));
```



```java
public void forEach(Consumer<? super E> action)
```

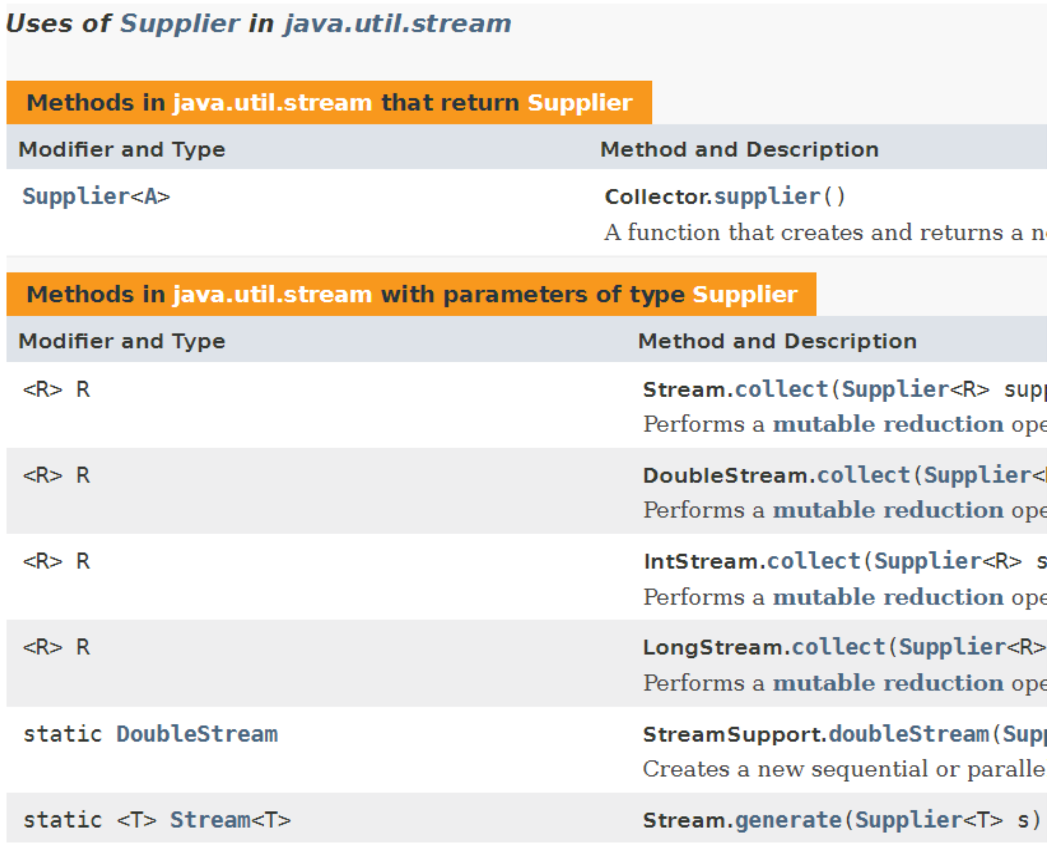
Supplier<T>

represents a function that takes no argument and returns (supplies) a value



```java
public class SupplierInterface {
    //Supplier function declarations.
    Supplier<String>  textSupplier     = () -> "Hello SW Test Academy!";
    Supplier<Integer> numberSupplier   = () -> 1234;
    Supplier<Double>  randomSupplier   = () -> Math.random();
    Supplier<Double>  randomSupplierMR = Math::random; //With Method Refe

    @Test
    public void supplierTest() {
        //Calling Supplier functions.
        System.out.println(textSupplier.get());
        System.out.println(numberSupplier.get());
        System.out.println(randomSupplier.get());
        System.out.println(randomSupplierMR.get());
    }
}
```

Example: https://www.swtestacademy.com/java-functional-interfaces/



Uses of Supplier in java.util.stream

Methods in java.util.stream that return Supplier

Modifier and Type	Method and Description
`Supplier<A>`	`Collector.supplier()` A function that creates and returns a n

Methods in java.util.stream with parameters of type Supplier

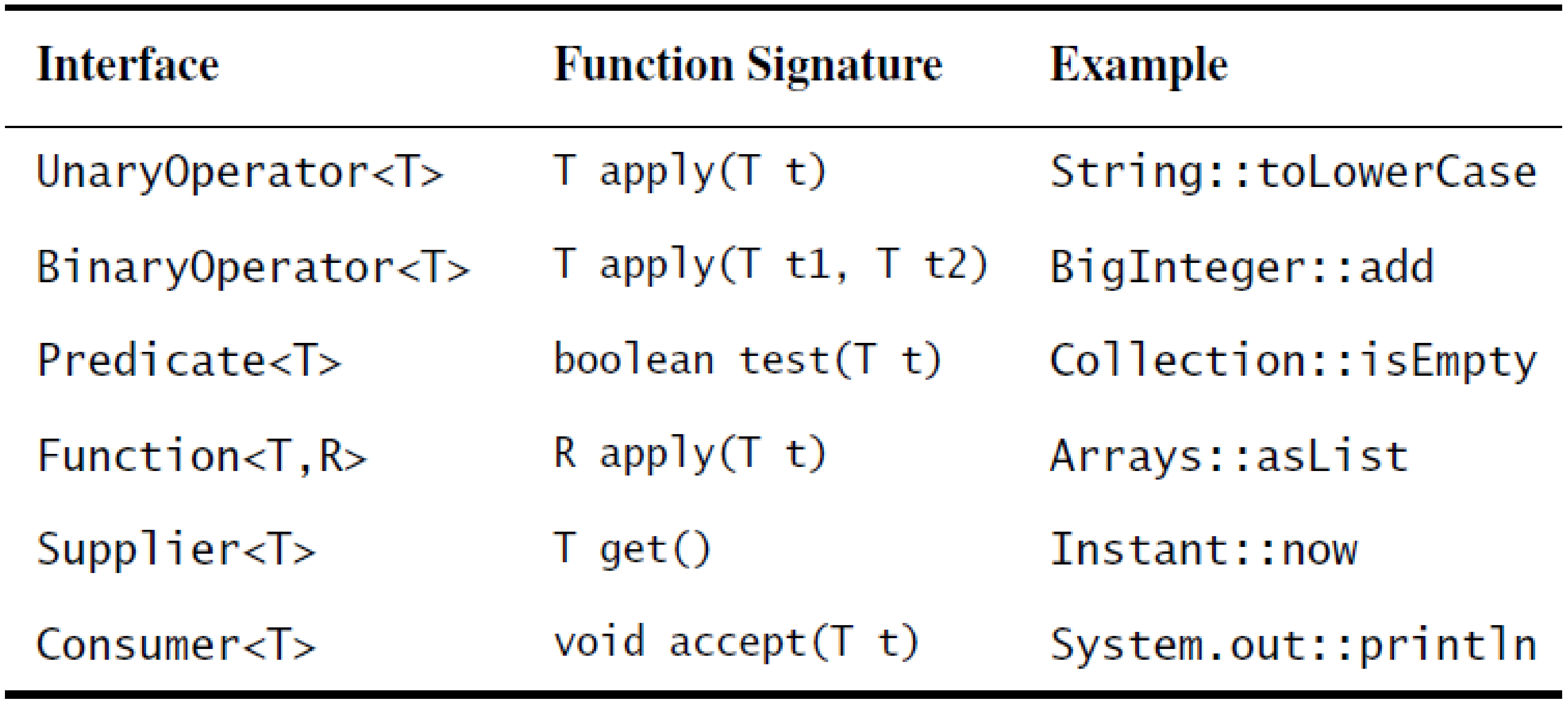
Modifier and Type	Method and Description
`<R> R`	`Stream.collect(Supplier<R> sup` Performs a **mutable reduction** ope
`<R> R`	`DoubleStream.collect(Supplier<` Performs a **mutable reduction** ope
`<R> R`	`IntStream.collect(Supplier<R> s` Performs a **mutable reduction** ope
`<R> R`	`LongStream.collect(Supplier<R>` Performs a **mutable reduction** ope
`static DoubleStream`	`StreamSupport.doubleStream(Sup` Creates a new sequential or paralle
`static <T> Stream<T>`	`Stream.generate(Supplier<T> s)`

Common Functional Interfaces

The `Operator` interfaces represent functions whose result and argument types are the same

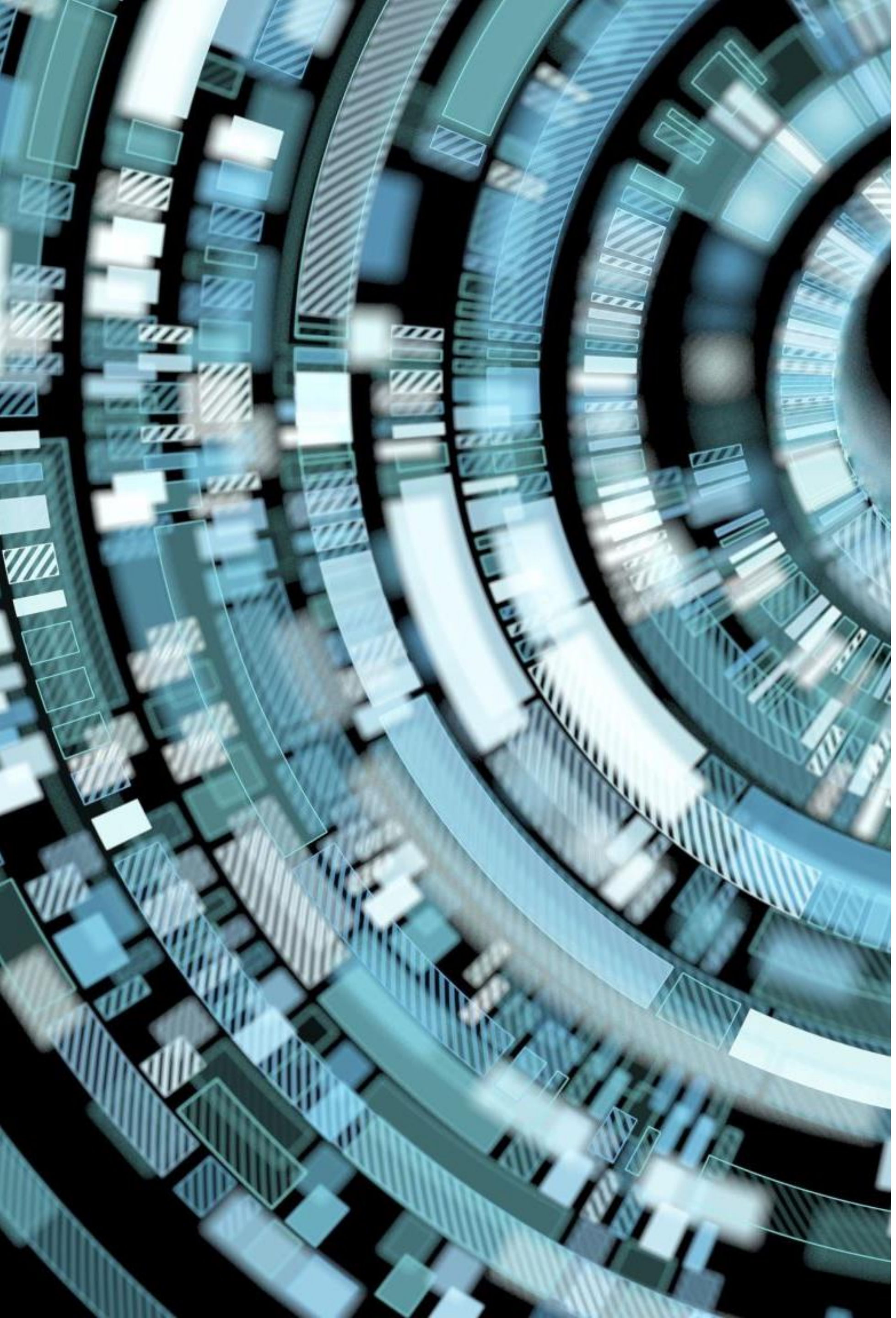
The `Predicate` interface represents functions who take an argument and return a boolean

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



Interface	**Function Signature**	**Example**
`UnaryOperator<T>`	`T apply(T t)`	`String::toLowerCase`
`BinaryOperator<T>`	`T apply(T t1, T t2)`	`BigInteger::add`
`Predicate<T>`	`boolean test(T t)`	`Collection::isEmpty`
`Function<T,R>`	`R apply(T t)`	`Arrays::asList`
`Supplier<T>`	`T get()`	`Instant::now`
`Consumer<T>`	`void accept(T t)`	`System.out::println`

Table from “Effective Java”



Lecture 4

- Functional Programming
- Lambda Expressions
- Stream API

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

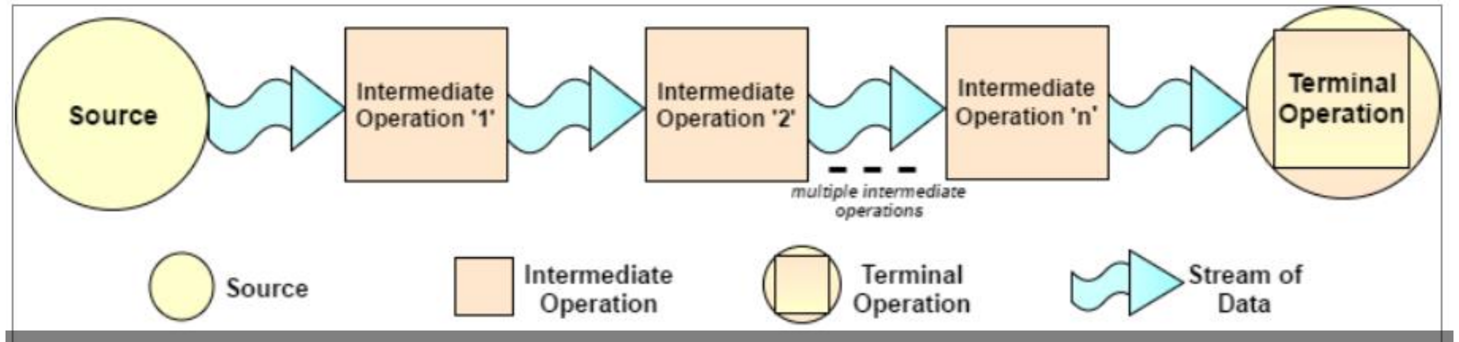
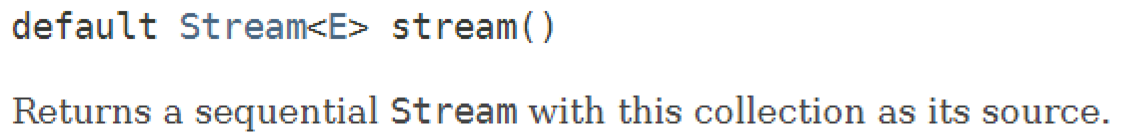

Source
Intermediate Operation '1'
Intermediate Operation '2'
Intermediate Operation 'n'
Terminal Operation
multiple intermediate operations
Source
Intermediate Operation
Terminal Operation
Stream of Data

Image from javabrahman.com

Create a Stream



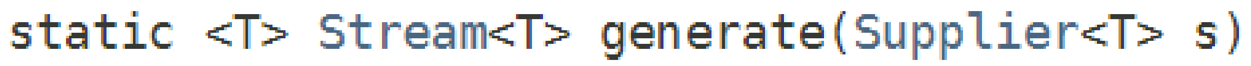
```
default Stream<E> stream()
```

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

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

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 <u>do not get executed</u> until a terminal operation is invoked (discussed later)

filter()

- 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()

- 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("abc");
strList.add("bcd");

strList.stream()
        .map(element -> element.toUpperCase())
        .forEach(element -> System.out.println(element));
```

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;
        }

}
```

```
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)) Get odd numbers: 1,3,5,7,9
       .limit(3)  Get first 3 odd numbers: 1,3,5
       .map(element -> (element*element)) Get the square: 1,9,25
       .forEach(element -> System.out.println(element));
```

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()

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

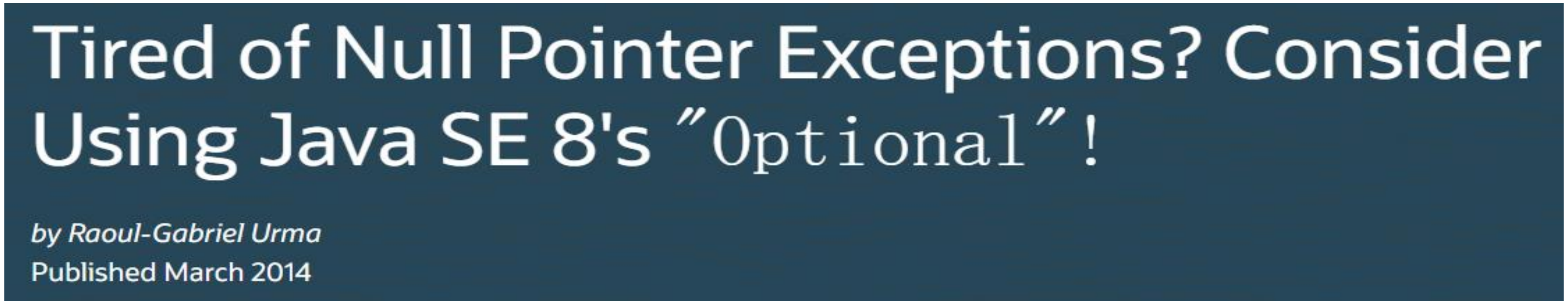
- Returns whether any elements of this stream match the provided predicate (check whether any element in list satisfies a given condition)

```
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.get());
```



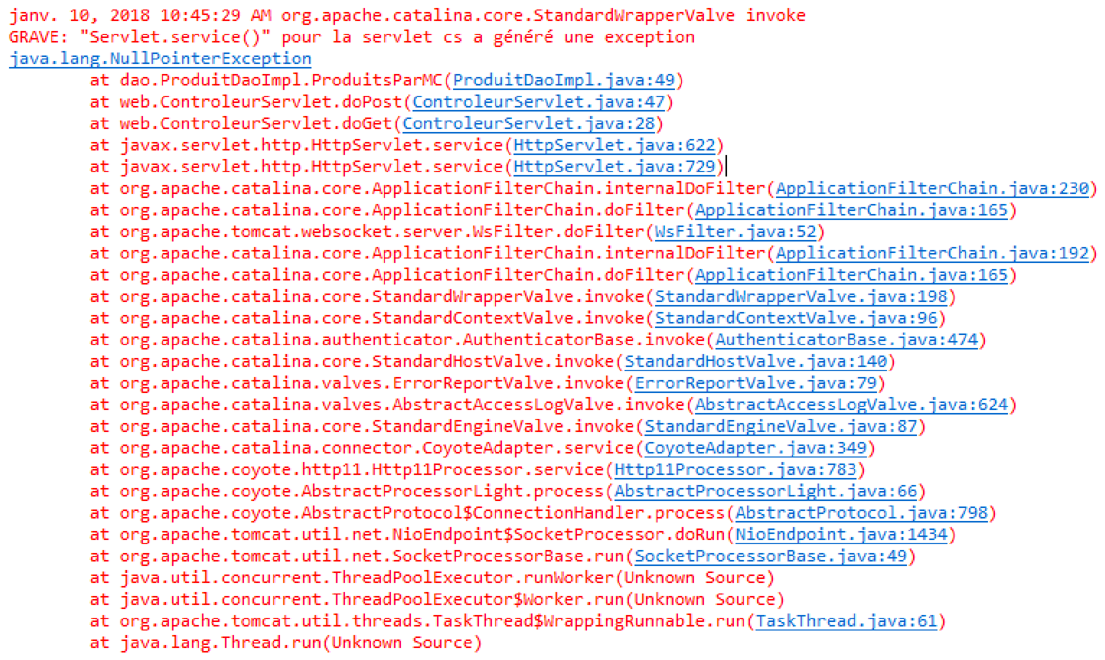
Tired of Null Pointer Exceptions? Consider Using Java SE 8's "Optional"!

by Raoul-Gabriel Urma

Published March 2014

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

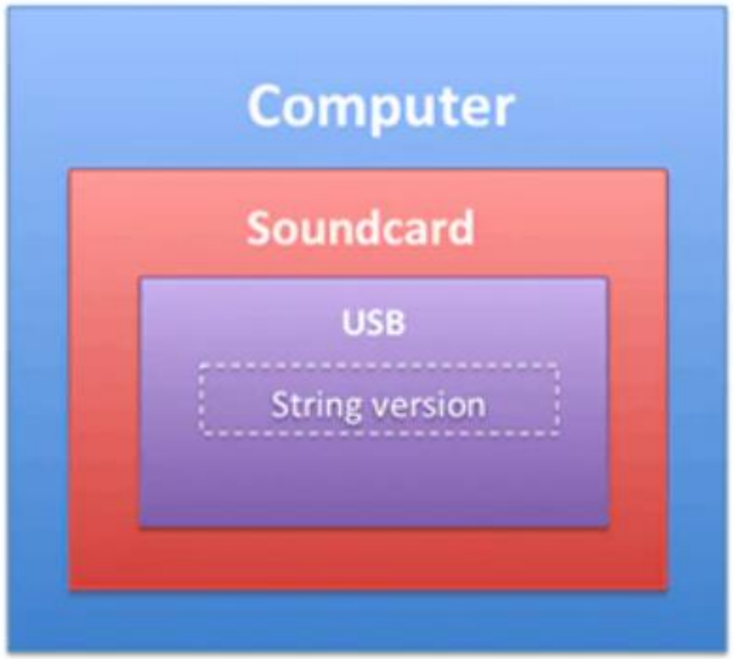
SIMPLY EXPLAINED
ARE YOU REALLY SURE THAT THIS VARIABLE CAN NEVER EVER BE NULL?
OF COURSE!!!
NullPointerException



```
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!



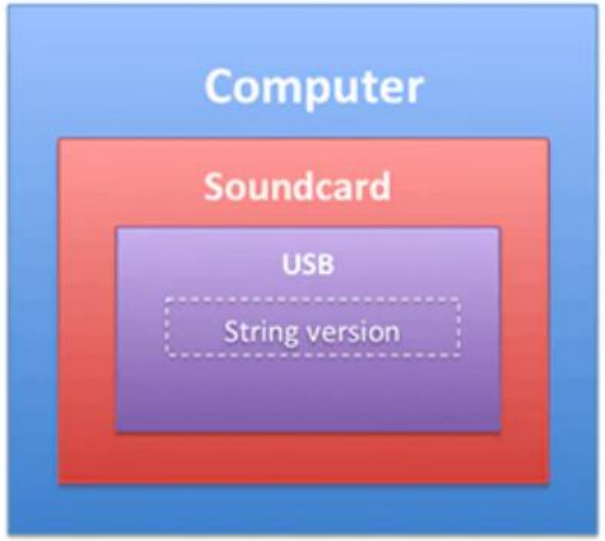
```
String version = "UNKNOWN";
if(computer != null){
  Soundcard soundcard = computer.getSoundcard();
  if(soundcard != null){
    USB usb = soundcard.getUSB();
    if(usb != null){
      version = usb.getVersion();
    }
  }
}
```

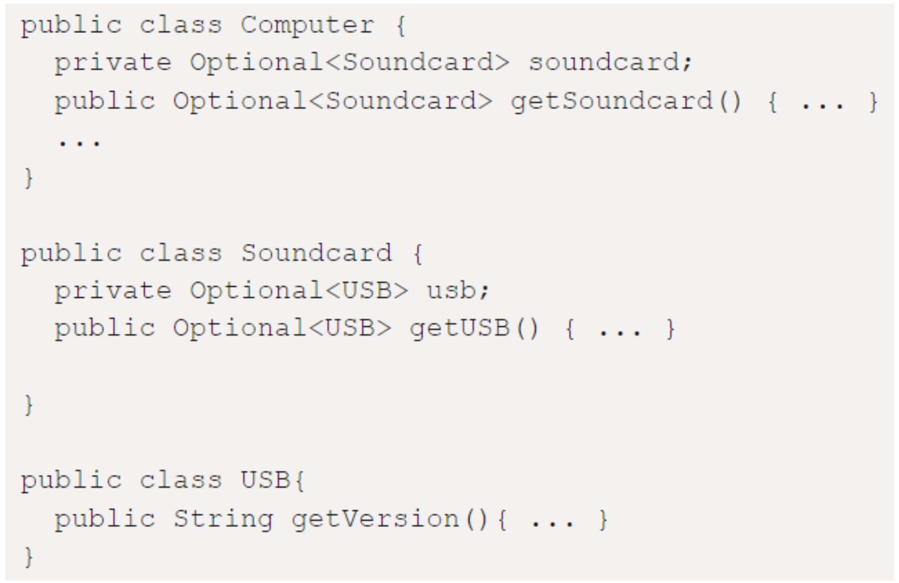
```
String version = computer.getSoundcard().getUSB().getVersion();
```

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
- Help us to specify alternative values to return or alternative actions to take if the value is `null`, without having to use null checkers





```
public class Computer {
  private Optional<Soundcard> soundcard;
  public Optional<Soundcard> getSoundcard() { ... }
  ...
}

public class Soundcard {
  private Optional<USB> usb;
  public Optional<USB> getUSB() { ... }

}

public class USB{
  public String getVersion(){ ... }
}
```

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

The Optional<T> class

- We can specify alternative values to return or alternative actions to take if the value is `null`, without having to use null checkers

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

```
String name = computer.flatMap(Computer::getSoundcard)
                          .flatMap(Soundcard::getUSB)
                          .map(USB::getVersion)
                          .orElse("UNKNOWN");
```

These are methods of the Optional class, not Stream!

Further reading:

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

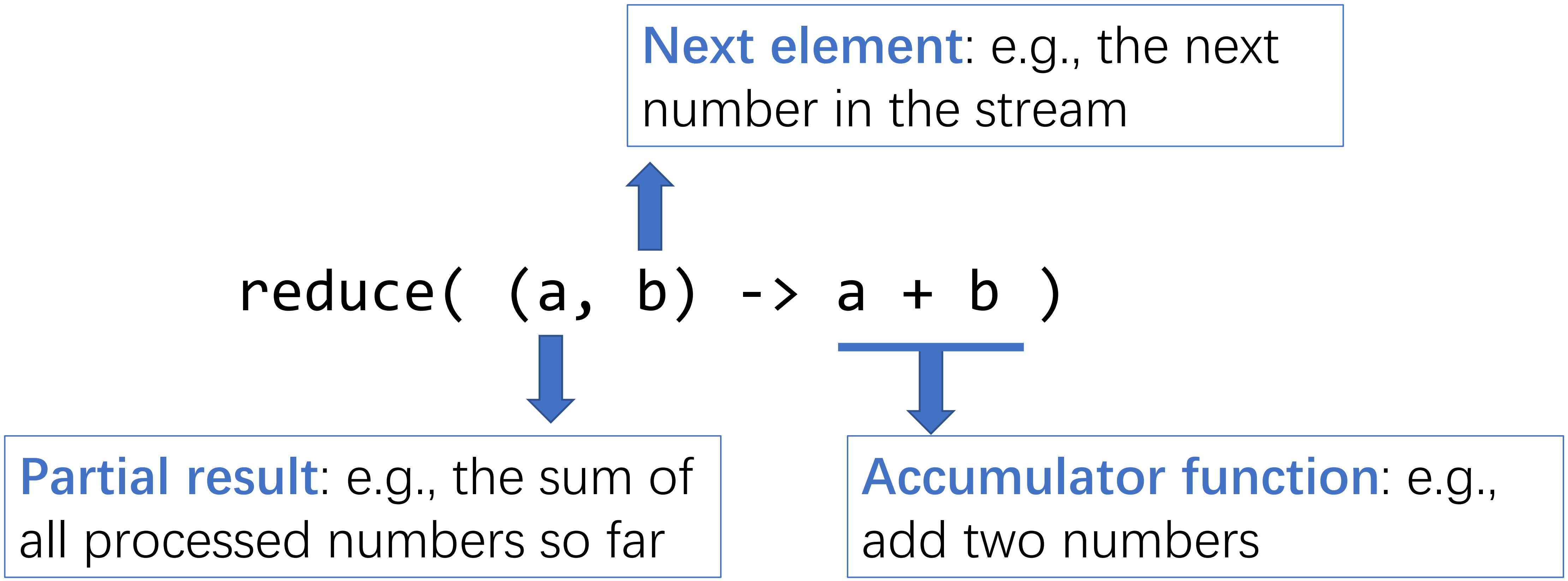
Chapter 9, "Optional: a better alternative to null," from *Java 8 in Action: Lambdas, Streams, and Functional-style Programming*

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()

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



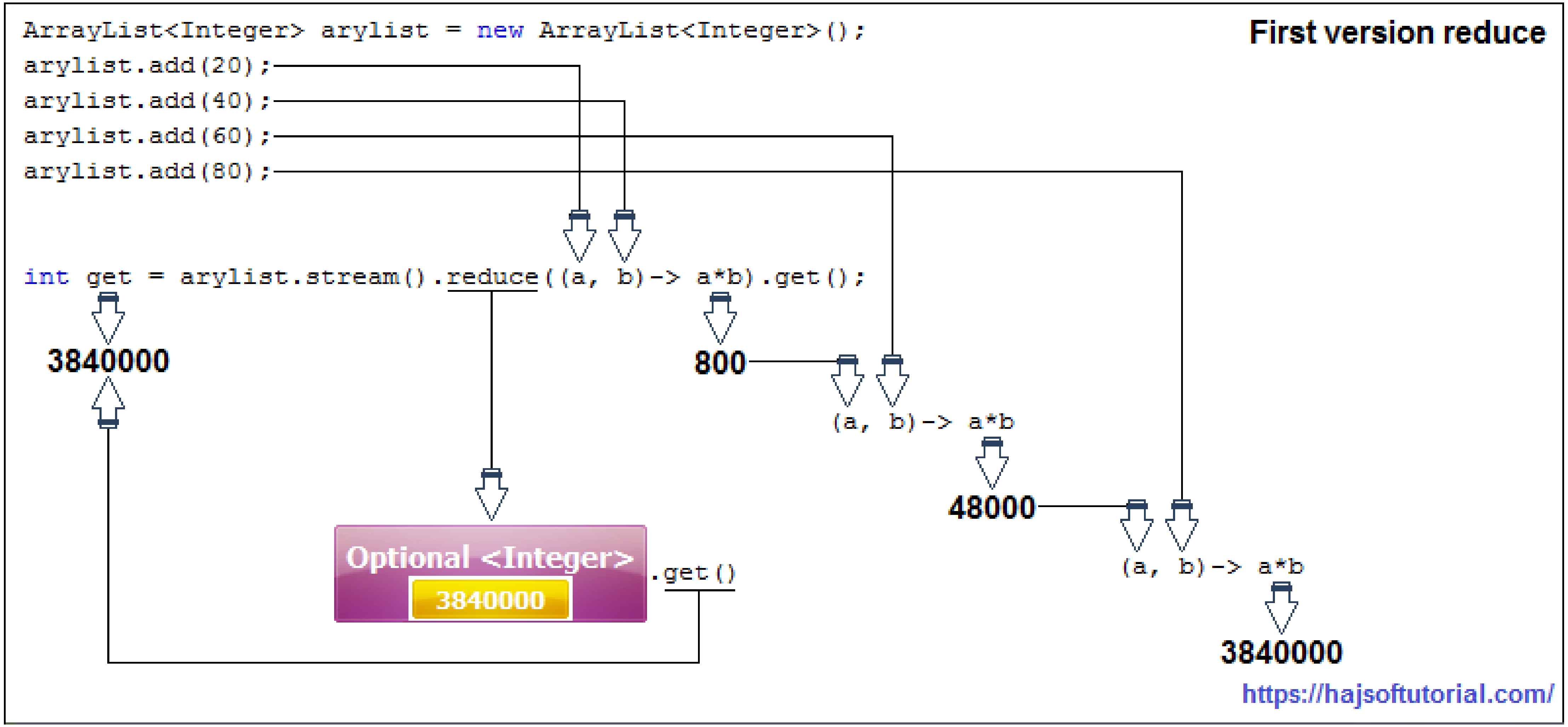
Next element: e.g., the next number in the stream

```
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



```
ArrayList<Integer> arylist = new ArrayList<Integer>();
arylist.add(20);
arylist.add(40);
arylist.add(60);
arylist.add(80);

int get = arylist.stream().reduce((a, b)-> a*b).get();
```

reduce() Example II

What's the output?

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

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

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

Lazy Evaluation

Benefits?

Intermediate operations are lazily executed

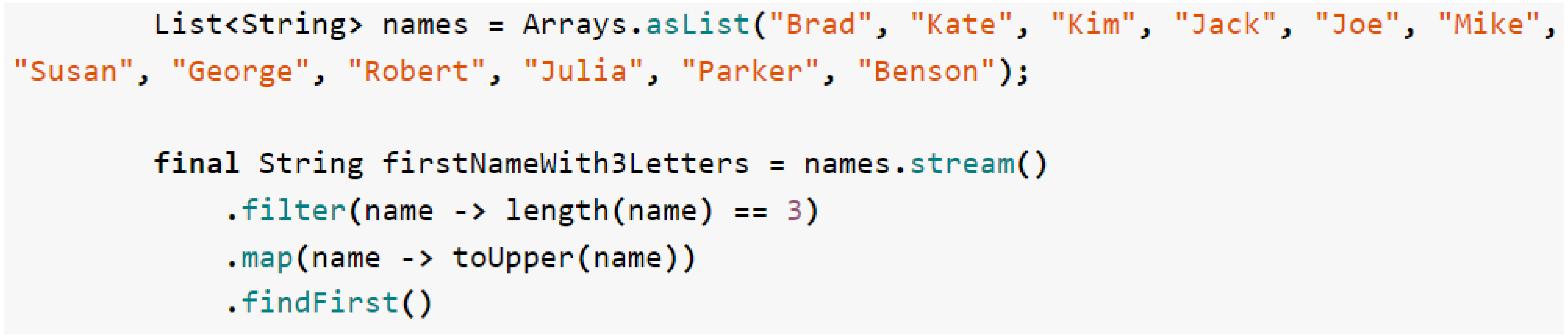
- They only remember the operations, but don't do anything right away (lazy)

Terminal operations are eagerly executed

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

Example

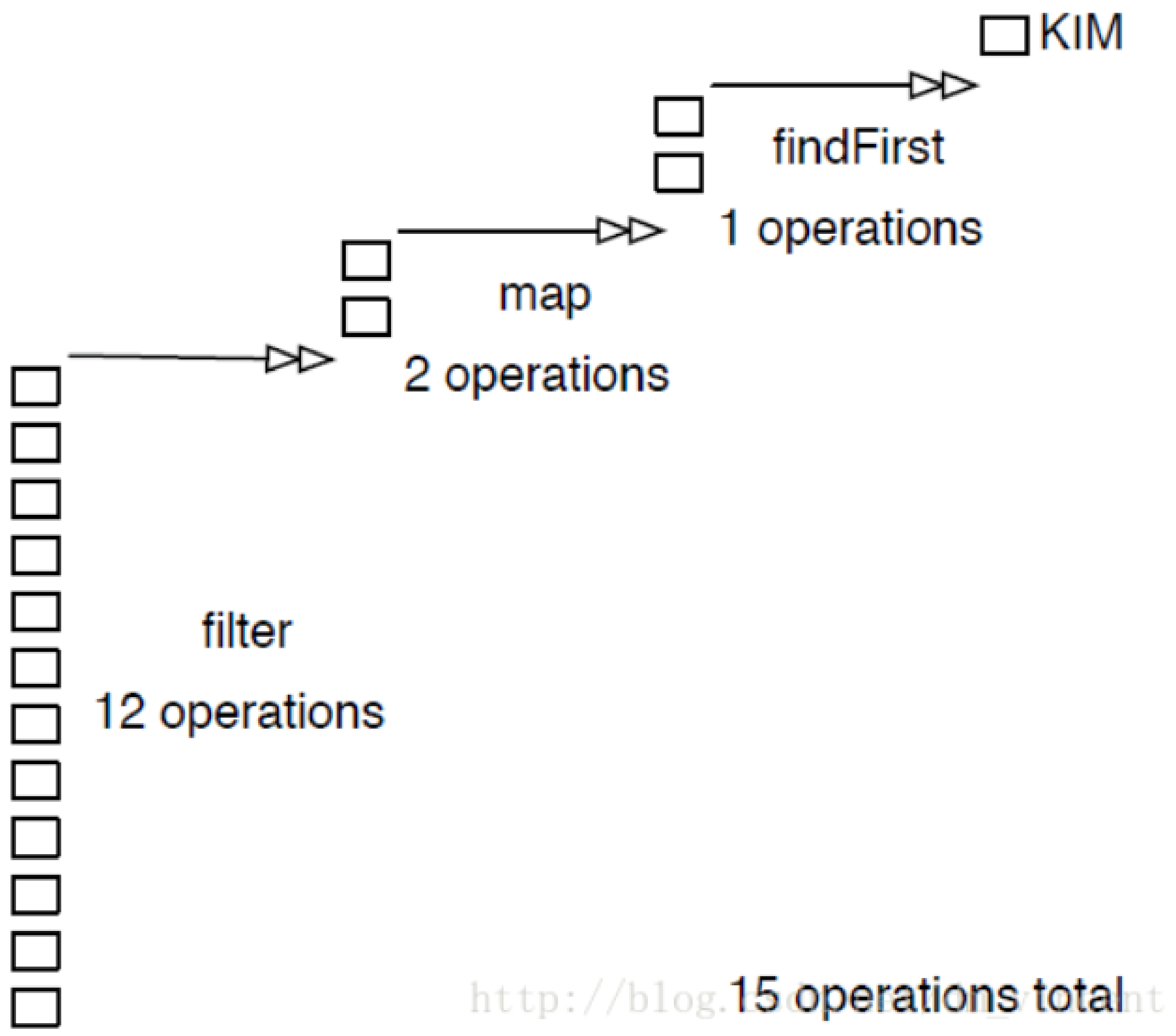
How many operations do we have to perform when the code is executed early / lazily?



```
        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()
```

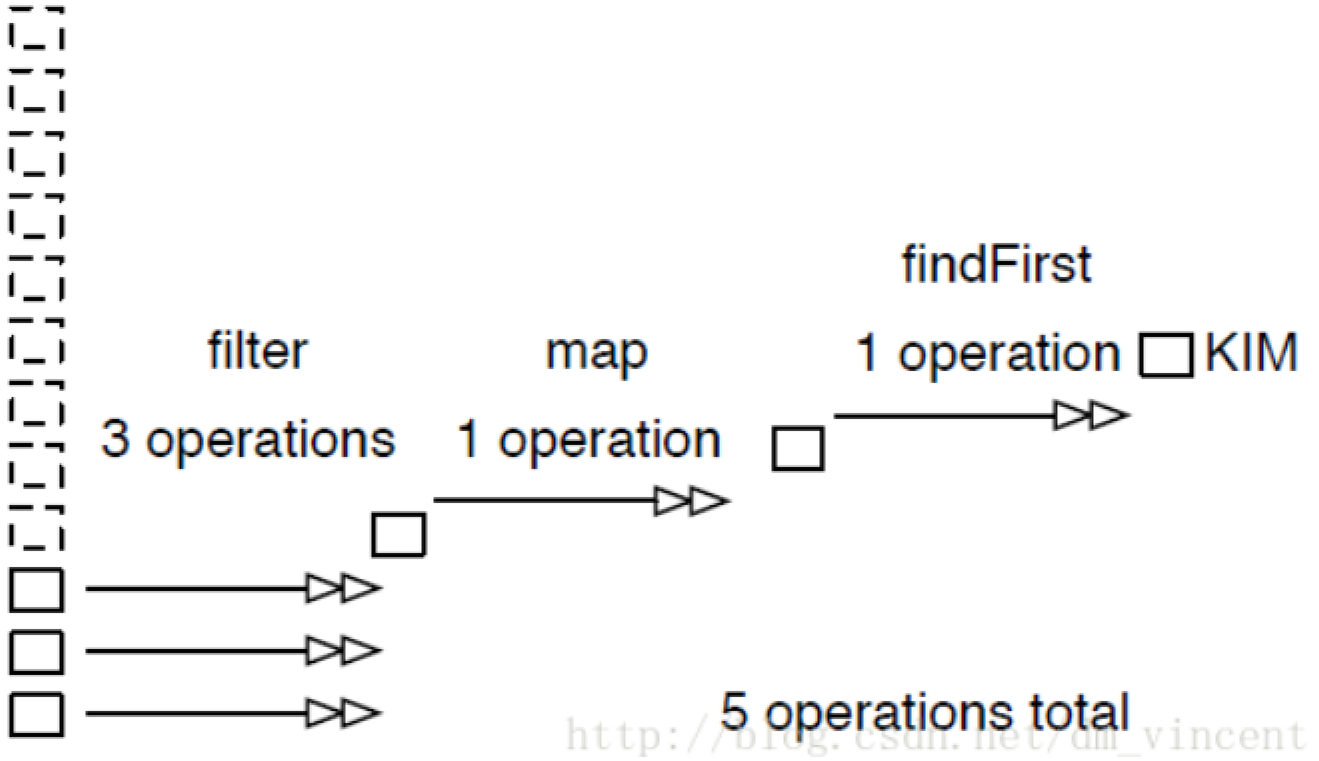
Reference: https://blog.csdn.net/dm_vincent/article/details/40503685

Eager Execution



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

Lazy Execution



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

Next Lecture

- Introduction to GUI
- Java FX