Stream API, Error handling

Working with the Stream API, (try/catch/finally), Checked / Unchecked Exceptions

```
function addNumbers(a, b) {
  return a + b;
};

// Takes the values of an array and returns the total. Demonstrates simple
// recursion.
function totalForArray(arr, currentTotal) {
  currentTotal = addNumbers(currentTotal + arr.shift());

  if(arr.length > 0) {
    return totalForArray(currentTotal, arr);
  }
  else {
    return currentTotal;
  }
}

// Or you could just use reduce.
function totalForArray(arr) {
  return arr.reduce(addNumbers);
}

// Should really be called divideTwoNumbers
function average(total, count) {
  return count / total;
}

function averageForArray(arr) {
  return average(arr.length, totalForArray(arr));
}

// Gets the value associated with the property of an object. Intended for
// use with a collection method like map, hence the generator.
function getItem(propertyName) {
  return function(item) {
    return item[propertyName];
  }
}
```

SoftUni Team

Technical Trainers

Software University

http://softuni.bg

Table of Contents

1. Stream<T> Class
2. Types of Streams in Stream API
   - Generic, Primitive
3. Types of Stream Operations
   - Intermediate, Terminal
   - Map, Filter, Reduce
4. Collectors
5. Streams on Maps
6. Error handling try/catch/finally
7. Checked / Unchecked Exceptions

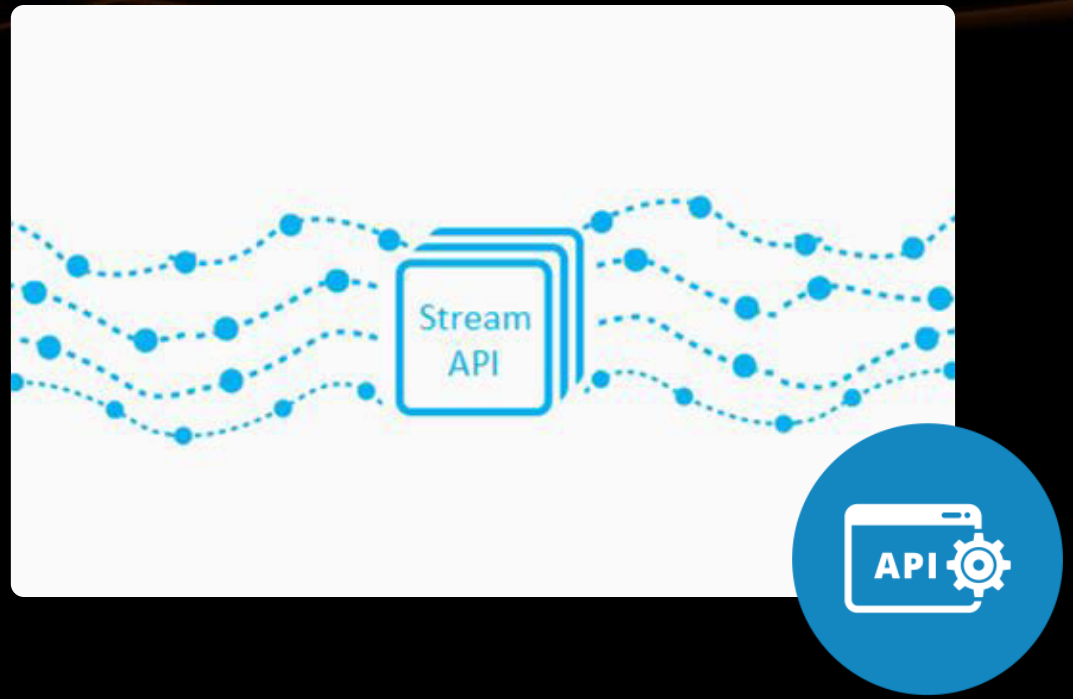
Questions

SOFTWARE UNIVERSITY FOUNDATION

sli.do

#JavaDB

SOFTWARE UNIVERSITY FOUNDATION

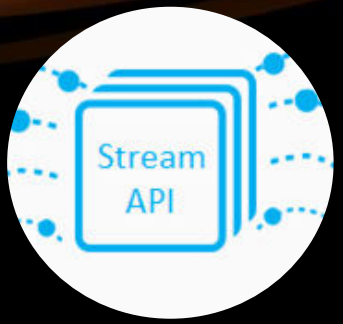


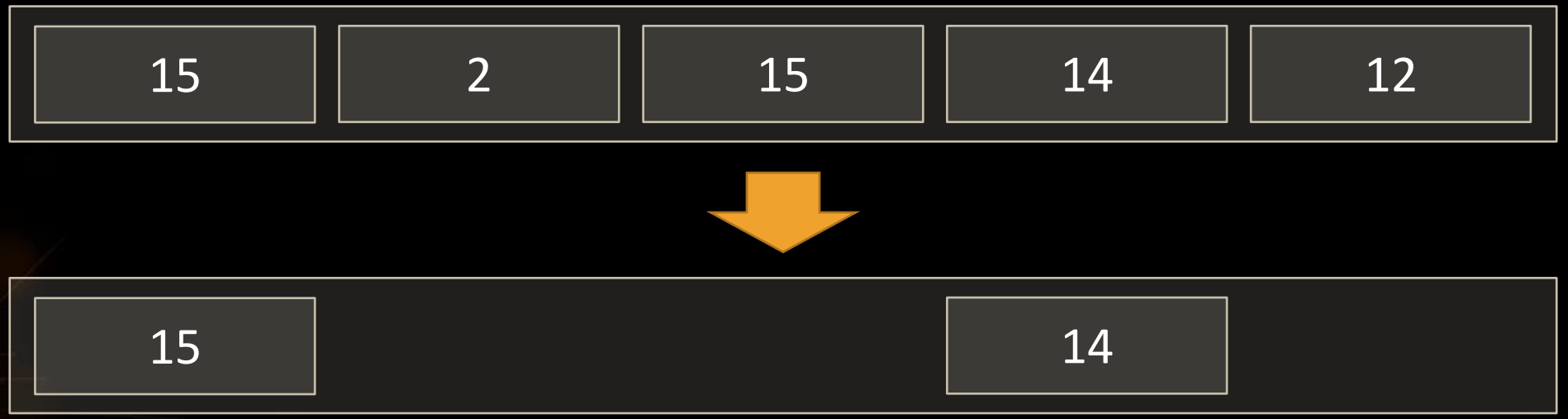
Stream API

Traversing and Querying Collections

The Stream API

- Querying a collection in a functional way
- Methods are chained together
- Example:
  - Find all unique elements, such that $10 \leq n \leq 20$ and take the first 2





Problem: Take Two

- Create a program that:
  - Reads a sequence of integers
  - Finds all unique elements, such that $10 \leq n \leq 20$
  - Prints only the first 2 elements

`15 2 15 14 12`	`15 14`
`17 -2 3`	`17`
`-2 3`	*`(no output)`*

Check your solution here: https://judge.softuni.bg/Contests/Practice/Index/465#0

Solution: Take Two – Non Functional

```
LinkedHashSet<Integer> set = // create set

for (Integer number : numbers) {
  if (set.size() >= 2) {
    break;
  }
  if (10 <= number && number <= 20) {
    set.add(number);
}}
```

Check your solution here: https://judge.softuni.bg/Contests/Practice/Index/465#0

Solution: Take Two – Functional

```
numbers.stream()
   .filter(n -> 10 <= n && n <= 20)
   .distinct()
   .limit(2)
   .forEach(n -> print(n));
```

Check your solution here: https://judge.softuni.bg/Contests/Practice/Index/465#0

Stream<T> Class

- Gives access to Stream API functions.
- Get an instance through:
  - A Collection:

```
List<Integer> list = new ArrayList<>();
Stream<Integer> stream = list.stream();
```

  - An Array:

```
String[] array = new String[10];
Stream<String> stream = Arrays.stream(array);
```

Stream<T> Class (2)

- Gives access to the Stream API.
- Get an instance through:
  - A Hash Map Collection:

```
HashMap<String, String> map = new HashMap<>();

Stream<Map.Entry<String, String>> entries =
        map.entrySet().stream();
Stream<String> keys = map.keySet().stream();
Stream<String> keys = map.values().stream();
```

Function Execution

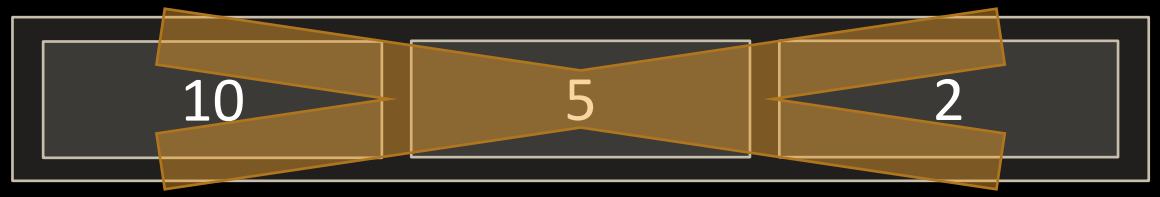
- Each function call creates a new Stream<T> instance
  - This allows method chaining

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

Stream<String> stringStream = strings.stream();

Stream<Integer> intStream =
        stringStream.map(s -> s.length());
```

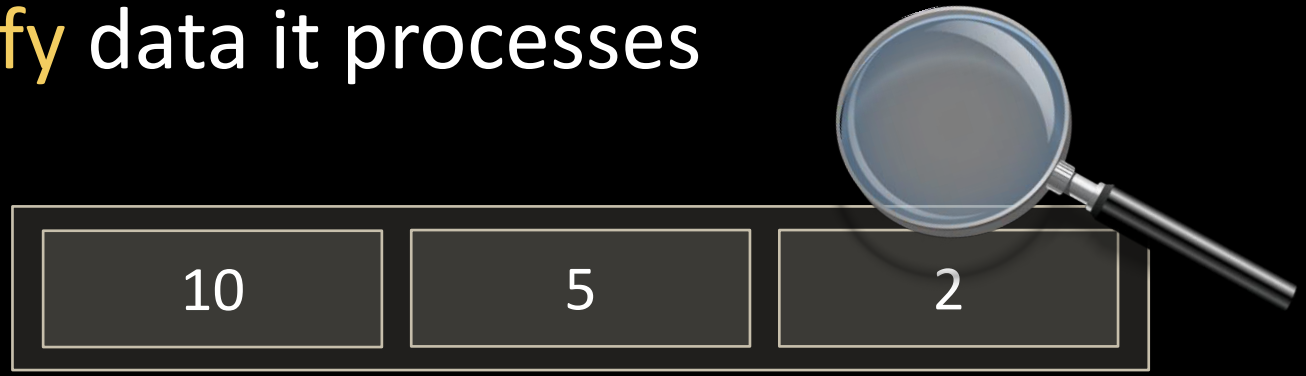
Function Execution (2)

So What is a Stream?

- Stream is not a collection and don't store any data



- Stream iterates over a collection
- Does not modify data it processes



Stream Types and Optionals

Generic and Primitive Streams

Generic Streams

- Can be of any type except primitives

```
List<String> strings = new ArrayList<>();
Stream<String> strStream = strings.stream();

List<Integer> ints = new ArrayList<>();
Stream<Integer> intStream = ints.stream();

List<Object> objects = new ArrayList<>();
Stream<Object> objStream = objects.stream();
```

Problem: UPPER STRINGS

- Read a sequence of strings
- Map each to upper case and print them
- Use the Stream API

```
Pesho Gosho Stefan
```

```
PESHO GOSHO STEFAN
```

```
Soft Uni Rocks
```

```
SOFT UNI ROCKS
```

Check your solution here: https://judge.softuni.bg/Contests/Practice/Index/465#0

Solution: UPPER STRINGS

```
Scanner scanner = new Scanner(System.in);
List<String> strings = Arrays.asList(
        scanner
        .nextLine()
        .split("\\s+"));

strings.stream()
   .map(s -> s.toUpperCase())
   .forEach(s -> System.out.print(s + " "));
```

Check your solution here: https://judge.softuni.bg/Contests/Practice/Index/465#0

Optional<T>

- Some functions can return Optional<T>

```
Optional<String> first = elements.stream()
        .sorted()
        .findFirst();

if (first.isPresent()) {
  System.out.println(first.get());
else
  System.out.println("No matches.");
```

Check if optional has value

Gets the value

Problem: First Name

- Read a sequence of names
- Read a letter
- Of the names that start with the letter find the first name by lexicographical order

```
Rado Plamen Pesho
p
```

```
Pesho
```

```
Plamen Gosho Rado
s
```

```
No match
```

Solution: First Name

SOFTWARE UNIVERSITY FOUNDATION

```
Scanner scanner = new Scanner(System.in);
List<String> names Arrays.asList(scanner.nextLine()
.split("\\s+"));

Character ch = scanner.nextLine().toLowerCase().charAt(0);

Optional<String> first = names.stream()
  .filter(name -> name.toLowerCase().charAt(0) ==
ch).sorted().findFirst();

if (first.isPresent())
  System.out.println(first.get());
else
  System.out.println("No match");
```

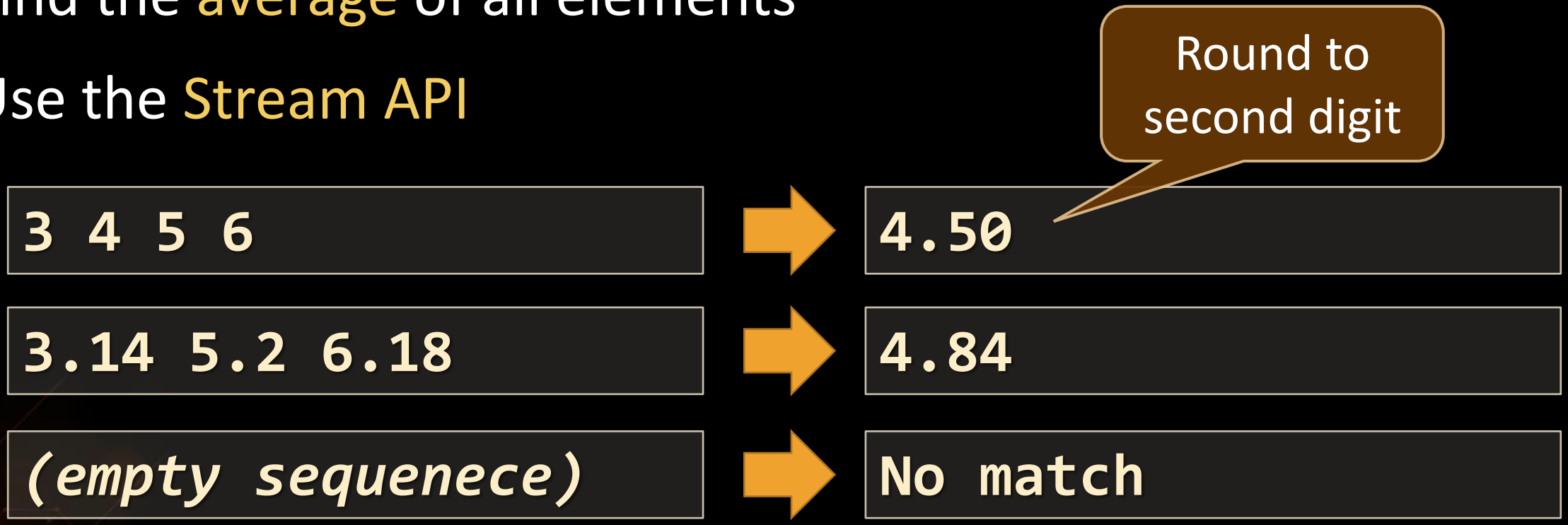
Primitive Streams

- Work efficiently with primitive types
- Give access to additional functions

```
int[] ints = { 1, 2, 3, 4 };
IntStream intStream = IntStream.of(ints);

List<Integer> list = new ArrayList<>();
IntStream mappedIntStream = list.stream()
        .mapToInt(n -> Integer.valueOf(n));
```

Problem: Average of Doubles

- Read a sequence of double numbers
- Find the average of all elements
- Use the Stream API



Check your solution here: https://judge.softuni.bg/Contests/Practice/Index/465#0

Solution: Average of Doubles

```
OptionalDouble average = elements.stream()
        .fiter(n -> !n.isEmpty())
        .mapToDouble(Double::valueOf)
        .average();

if (average.isPresent())
   System.out.printf(
        "%.2f", average.getAsDouble());
else
   System.out.println("No match");
```

Gets the value

Check your solution here: https://judge.softuni.bg/Contests/Practice/Index/465#0

Types of Operations

Intermediate, Terminal

Intermediate Operations

- Does not terminate the Stream

```
List<String> elements = new ArrayList<>();
Collections.addAll(elements, "one", "two");
Stream<String> stream = elements.stream()
        .distinct()
        .sorted()
        .filter(s -> s.length() < 5)
        .skip(1)
        .limit(1);
```

All return a new Stream

Allows function chaining

Intermediate Operations (2)

- Some of the intermediate operations

Function	Preserves count	Preserves type	Preserves order
map	✓	✗	✓
filter	✗	✓	✓
distinct	✗	✓	✓
sorted	✓	✓	✗
peek	✓	✓	✓

Terminal Operations

- Terminates the stream

```
List<String> elements = new ArrayList<>();
Collections.addAll(elements, "one", "two");

elements.stream()
        .distinct()
        .forEach(s -> System.out.println(s))
```

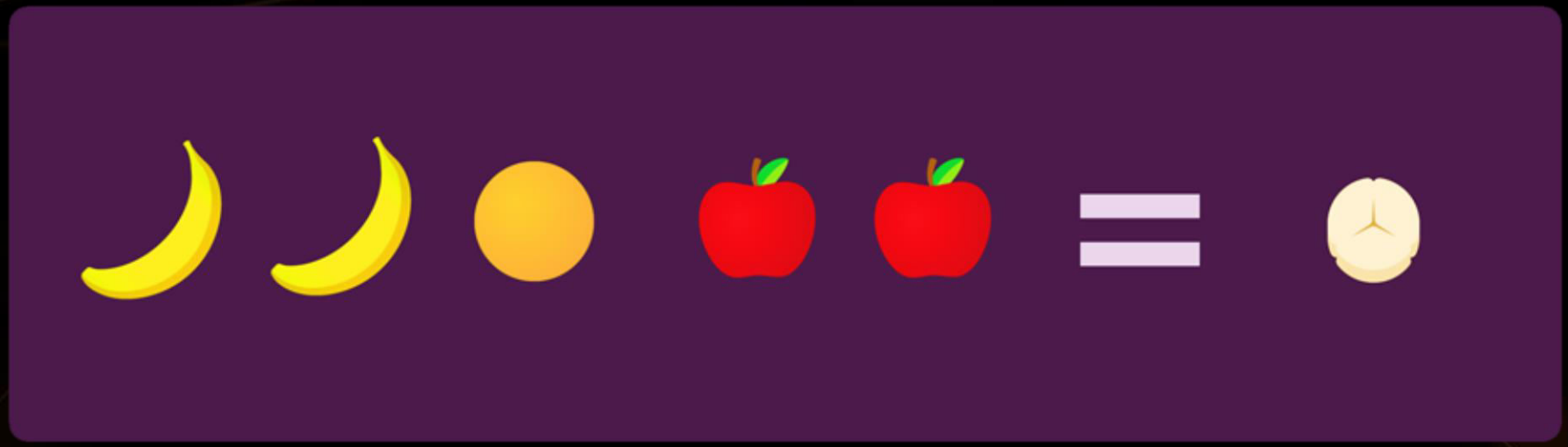
Closes the stream

Terminal Operations (2)

- Useful terminal operations:

Function	Output	When to use
reduce	concrete type	to cumulate elements
collect	list, map or set	to group elements
forEach	side effect	to perform a side effect on elements

SOFTWARE UNIVERSITY FOUNDATION

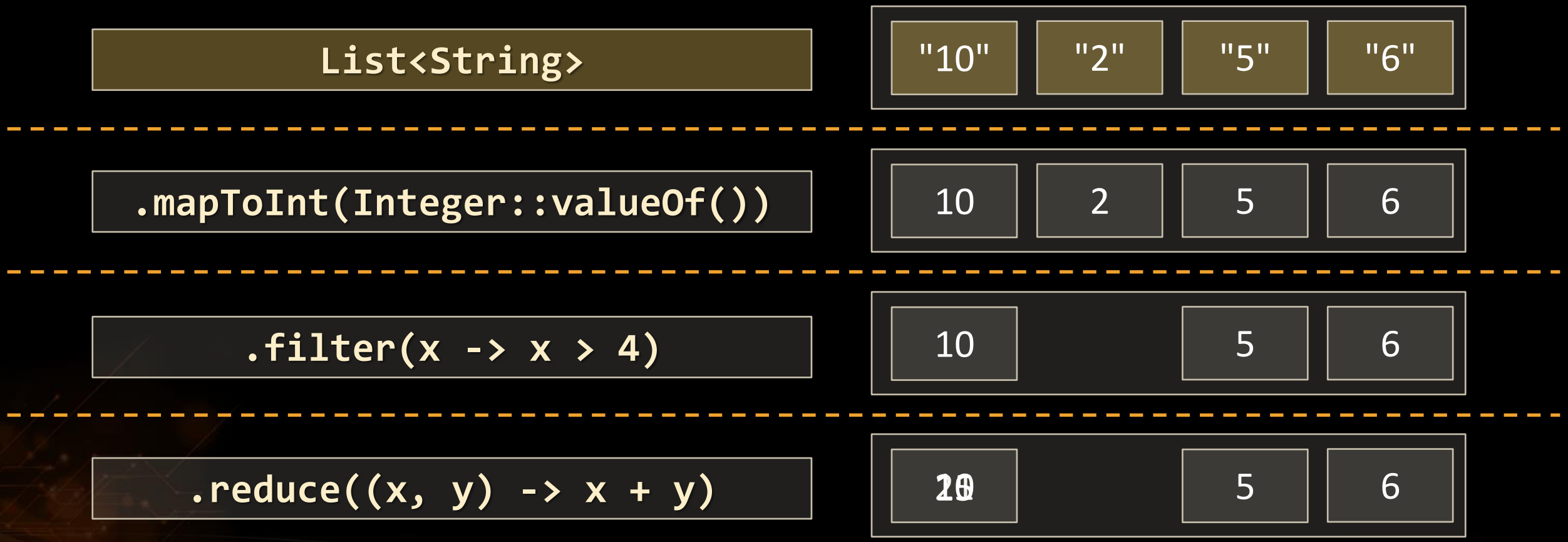


Types of Operations

Map, Filter, Reduce

Map, Filter, Reduce

- Common pattern in data querying



Map Operations

- Transform the objects in the stream

```
Stream<String> strStream =
      Stream.of("1", "2", "3");

Stream<Integer> numStream =
      strStream.map(Integer::valueOf);
```

Transforms the stream

Filter Operations

- Filters objects by a given predicate

```
Stream<String> strStream =
Stream.of("one", "two", "three")
      .filter(s -> s.length() > 3);
```

Preserves strings longer that 3

Reduce Operations

- Check for a given condition:
  - Any element matches:

```
boolean any = stream1.anyMatch(x -> x % 2 == 0);
```

  - All elements match:

```
boolean all = stream2.allMatch(x -> x % 2 == 0);
```

  - None of the elements match:

```
boolean none = stream3.noneMatch(x -> x % 2 == 0);
```

Short circuit operations

Find Reductions

- Find an element:
  - Gets the first element of the stream:

```
Optional<Integer> first = list.stream()
        .findFirst();
```

  - Gets any element of the stream:

```
Optional<Integer> first = list.stream()
        .findAny();
```

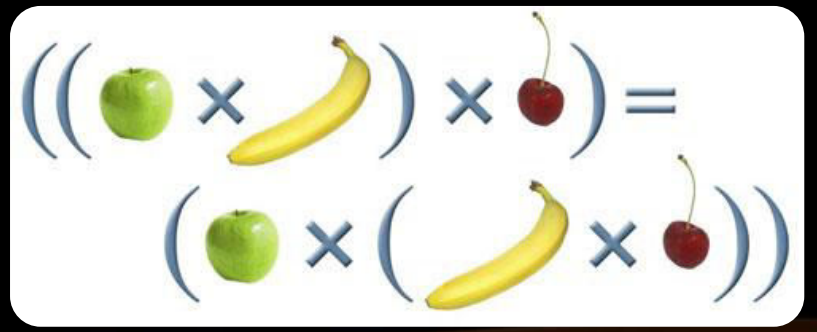
General Reduction

- Applies a given lambda:

```
Optional<Integer> first = list.stream()
        .reduce((x, y) -> x + y);
```

- Consider associativity:

```
r(a, r(b, c)) should be equal to r(r(a, b), c)
```



Problem: Min Even Number

- Read a sequence of numbers
- Find the min of all even numbers
- Use the Stream API

`1 2 3 4 5 6`		`2.00`
`3.14 -2.00 1.33`		`-2.00`
`(empty list)`		`No match`

Check your solution here: https://judge.softuni.bg/Contests/Practice/Index/465#0

Solution: Min Even Number

```
Optional<Double> min =
      Arrays.stream(
            scanner.nextLine().split("\\s+"))
      .filter(n -> !n.isEmpty())
      .map(Double::valueOf)
      .filter(n -> n % 2 == 0)
      .min(Double::compare);
```

Check your solution here: https://judge.softuni.bg/Contests/Practice/Index/465#0

Problem: Find and Sum Integers

- Read a sequence of elements
- Find the sum of all integers
- Use the Stream API

`Sum 3 and 4`		`7`
`Sum -3 and -4`		`-7`
`Sum three and four`		`No match`

Check your solution here: https://judge.softuni.bg/Contests/Practice/Index/465#0

Solution: Find and Sum Integers

```
Optional<Integer> sumOfIntsGT20 =
      Arrays.stream(
            scanner.nextLine().split("\\s+"))
      .filter(x -> isNumber(x))
      .map(Integer::valueOf)
      .reduce((x1, x2) -> x1 + x2);
```

Implement boolean method

Check your solution here: https://judge.softuni.bg/Contests/Practice/Index/465#0

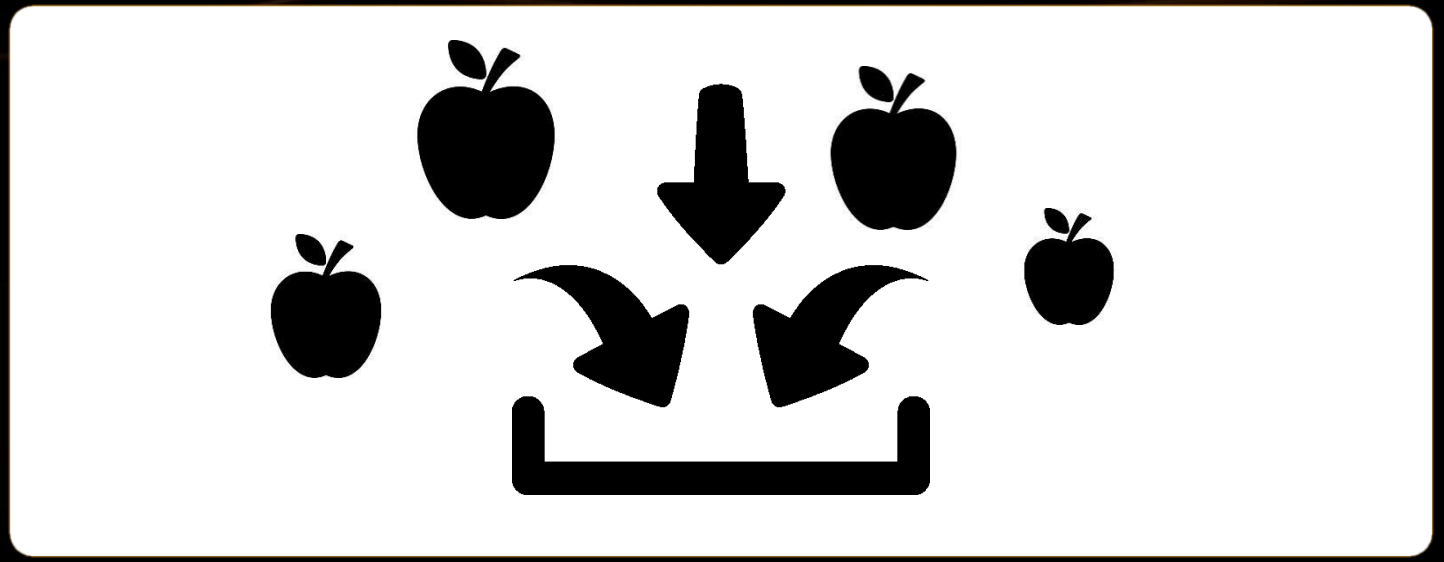
Sorting

- Sort by passing a comparator as lambda:

```
List<Integer> numbers = new ArrayList<>();
Collections.addAll(numbers, 7, 6, 3, 4, 5);

numbers.stream()
  .sorted((x1, x2) -> Integer.compare(x1, x2))
  .forEach(System.out::println);
```

(x2, x1) for descending order

SOFTWARE UNIVERSITY FOUNDATION



Collectors

Materializing a Stream

Collectors

- Collecting a Stream into a list:

```
String[] strings = { "22", "11", "13" };
List<Integer> numbers = Arrays.stream(strings)
        .map(Integer::valueOf)
        .collect(Collectors.toList());
```

- You can collect streams into different collections:
  - Arrays, Set, Map, etc.

Problem: Bounded Numbers

- Read a lower and upper bound
- Read a sequence of numbers
- Print all numbers, such that lower bound ≤ n ≤ upper bound

```
5 7
1 2 3 4 5 6 7 8 9
```

```
5 6 7
```

```
7 5
9 5 7 2 6 8
```

```
5 7 6
```

Check your solution here: https://judge.softuni.bg/Contests/Practice/Index/465#0

Solution: Bounded Numbers

```
Scanner scanner = new Scanner(System.in);

List<Integer> bounds =
        Arrays.stream(scanner.nextLine().split("\\s+"))
  .map(Integer::valueOf)
  .collect(Collectors.toList());

// continues…
```

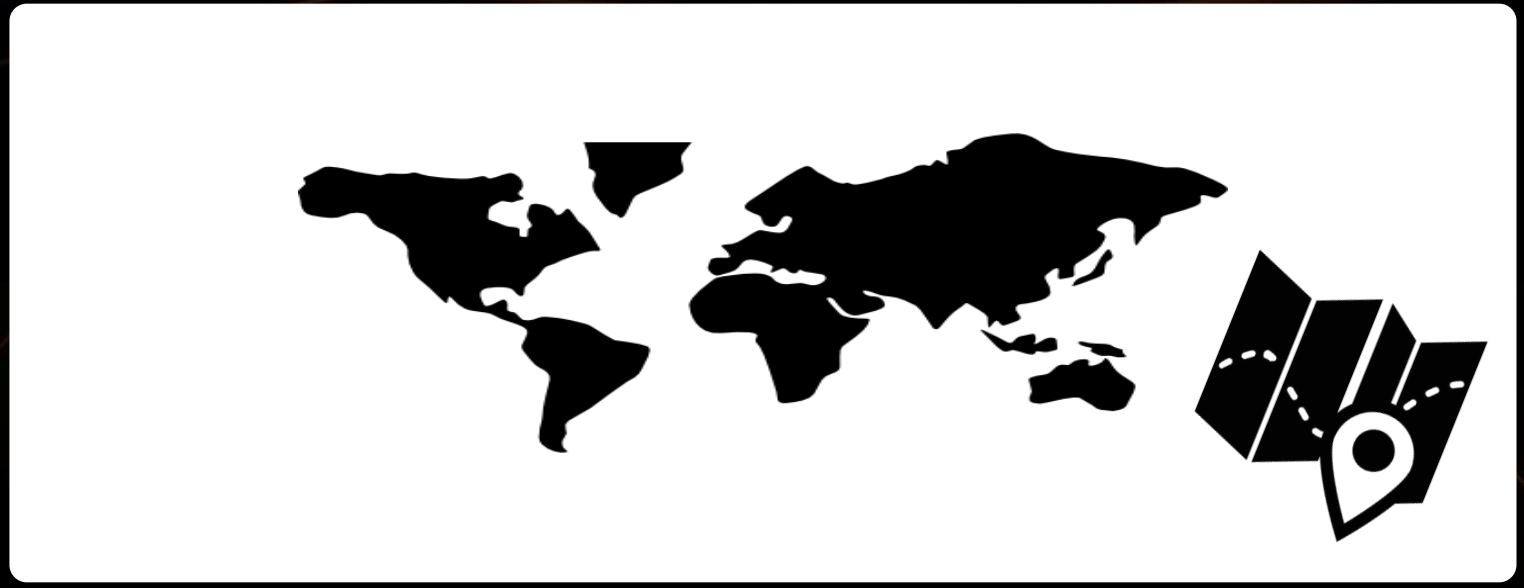
Check your solution here: https://judge.softuni.bg/Contests/Practice/Index/465#0

Solution: Bounded Numbers (2)

```
List<Integer> numbers =
        Arrays.stream(scanner.nextLine().split("\\s+"))
  .map(Integer::valueOf)
  .filter(x ->
        Collections.min(bounds) <= x
        && x <= Collections.max(bounds))
  .collect(Collectors.toList());
```

Check your solution here: https://judge.softuni.bg/Contests/Practice/Index/465#0

SOFTWARE UNIVERSITY FOUNDATION



Streams on Maps

Creating a Stream over a Map

Creating the Stream

- Use any dimension of the Hash Map:
  - Stream over the Entry set:

```
Stream<Map.Entry<String, String>> entries =
        map.entrySet().stream();
```

  - Stream over the Key set:

```
Stream<String> keys = map.keySet().stream();
```

  - Stream over the Value set:

```
Stream<String> values = map.values().stream();
```

Problem: Book Store

- Create class Book with name, author and price fields
- Create a steam of Books
- For each distinct author find the sum of prices of his books
- Sort result Map by decreasing sum of prices, then by author name alphabetically

```
Stream<Book> books = Stream.of(new Book("Vinetu3", 20, "Karl
Mai"),new Book("Vinetu1", 20, "Karl Mai"),
        new Book("Vinetu2", 15, "Karl Mai"),
        new Book("Sherlock Holmes", 12, "Arthur C. Doyl"),
        new Book("The Lost World", 43, "Arthur C. Doyl"));
```

```
Arthur C. Doyl=55.0
Karl Mai=55.0
```

Solution: Book Store

```
books.collect(
    Collectors.groupingBy(Book::getAuthor,
    Collectors.summingDouble(Book::getPrice))
    )
    .entrySet()
    .stream()
    .sorted(Map.Entry.<String, Double>comparingByValue()
    .reversed()
    .thenComparing(Map.Entry.<String,
Double>comparingByKey()))
    .forEach(System.out::println);
```

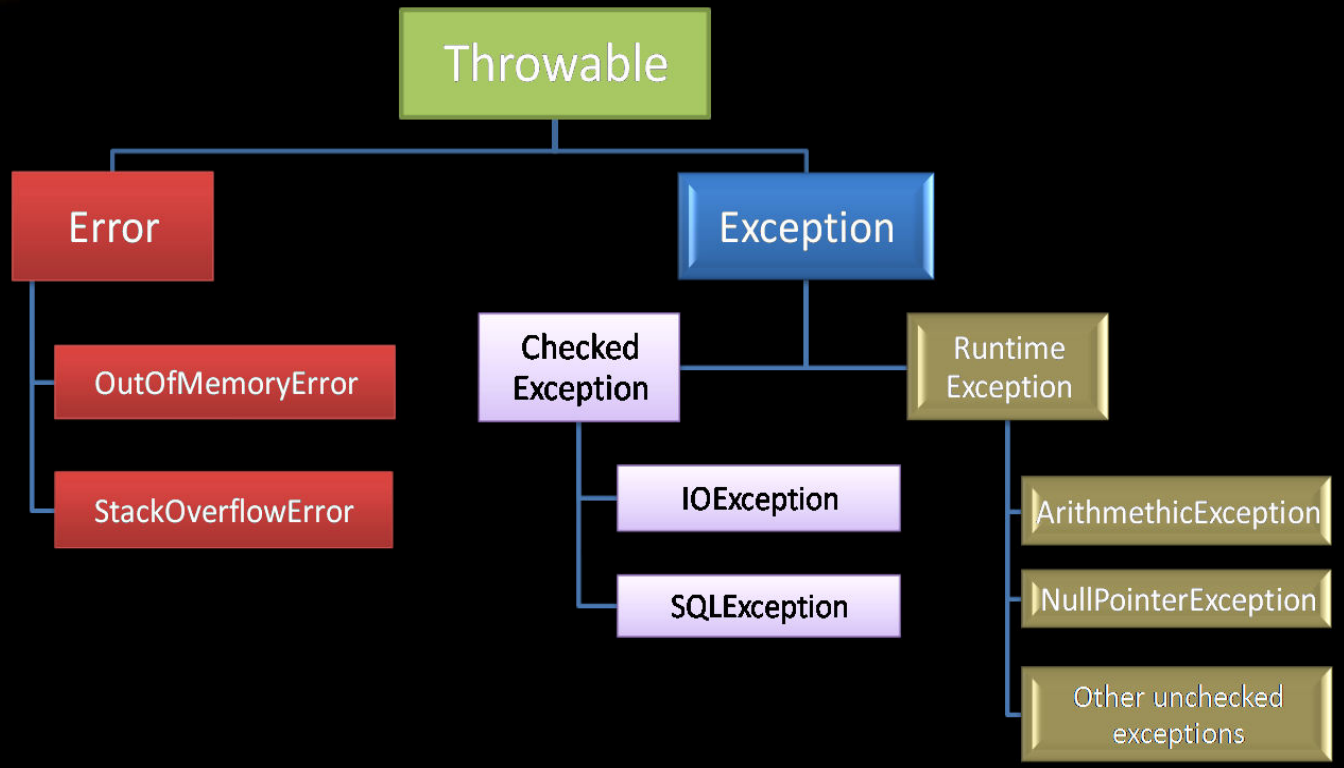
SOFTWARE UNIVERSITY FOUNDATION

Error Handling

Working with Exceptions

Exceptions in Java

- Exceptions are objects derived from Throwable interface



- Throwables are used to describe and handle failures at Runtime

Types of runtime failures

- Recoverable - Exception
  - Checked by the compiler - specify in method declaration by throws clause or catch it

```
public abstract int read() throws IOException;
```

  - Unchecked exceptions derived from RuntimeException, not obliged to declare or catch

```
public class NullPointerException extends RuntimeException
```

- Unrecoverable - Errors

Exceptions

- Exception object is created like any other object by calling new

```
public Exception (String message, Throwable cause)
```

- Its holding description of the failure in **message**

```
public String getMessage()
```

- It may wrap another Throwable object which is the root cause of the failure

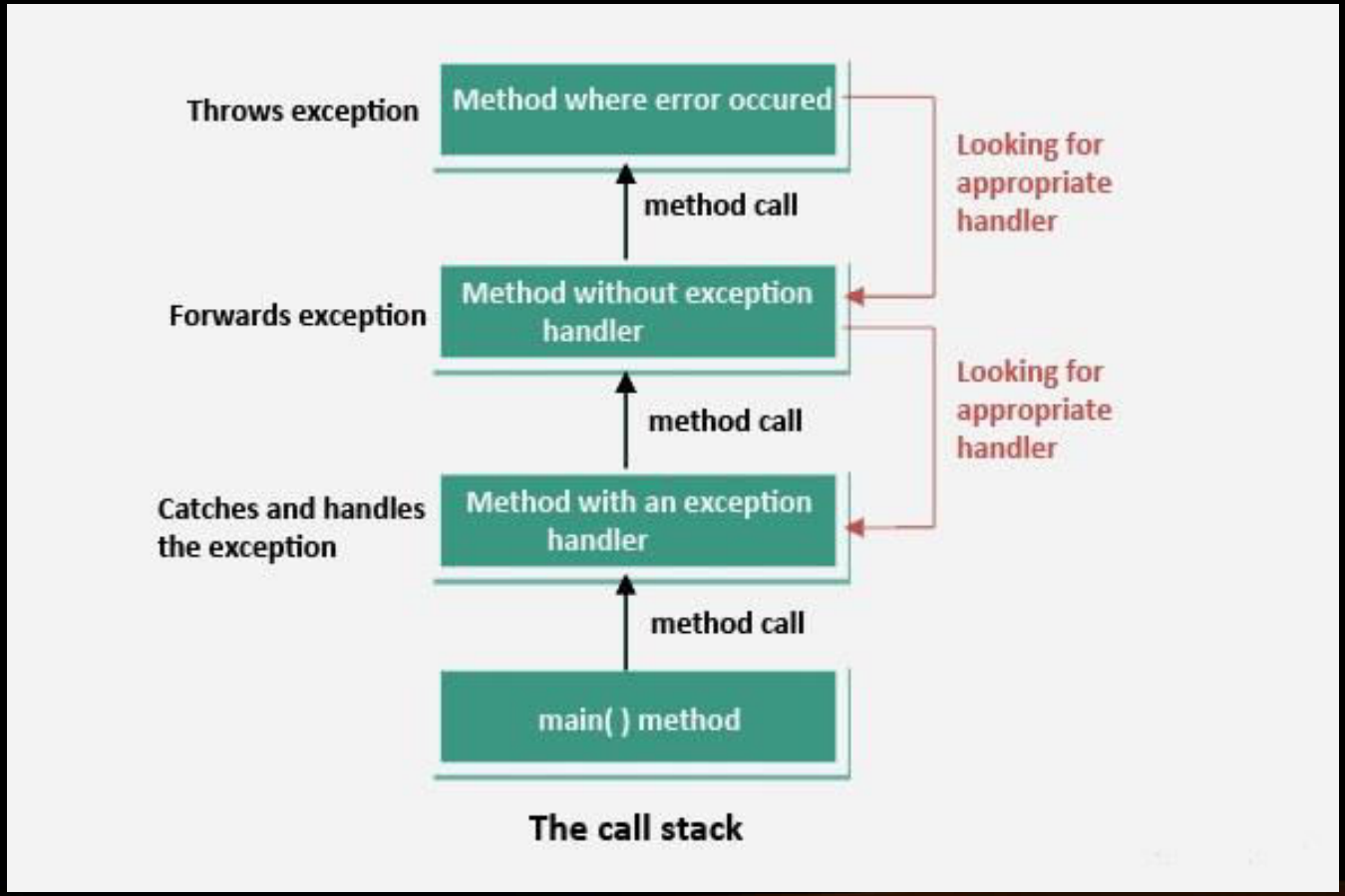
```
public Throwable getCause()
```

- It provides information about place of the failure

```
public void printStackTrace()
```

Exceptions and Call Stack

- An exception is an event (object), which is thrown during the execution of a program, that disrupts the normal flow of the program's instructions
- Execution goes backwards through the call stack until a proper handler is met
- In case no handler is met program terminates with an exception



Error Handling

```
//normal code execution which acquires some resource
try {
        // use resource, exceptions possible
}
catch (SomeException e) {
        //exception is caught, but what to do ???
        //Should I catch it if I do
        //not know how to recover from it ???
        //Recover OR Report and re-throw !!!
}
finally {
        //always executed, no matter if exception was
        //thrown or not
        //nice place to release the resource
}
```

Problem: Print File

- Write a program which takes a path to a file.
- The program should read the file and print its content to the console.
- Make sure the program is closing all resources it uses.
- Inform the user in case of file printed successfully.
- Inform the user for the case of reading operation failed.

Solution(1): Print File

```
private static void printFile(String fileName) throws
FileNotFoundException {
File sourceFile = new File(fileName);
Scanner = new Scanner(sourceFile);
String line = "";
try {
        while (scanner.hasNext()) {
                line = scanner.nextLine();
                System.out.println(line);
        }
} finally {
        scanner.close();
}}
```

Solution(2): Print File

```
//initialize Scanner, isPrinted and ch variables
try {
do {
        System.out.println("Enter path to the file to print : ");
        String filePath = scanner.nextLine();
        try {
                printFile(filePath);
                isPrinted = true;
        } catch (FileNotFoundException e) {
                System.out.println("File with name \"" + filePath
                + " \" is not found !");
                System.out.println("Do you want to try again?(y or n)");
                ch =  scanner.next().toLowerCase().trim().charAt(0);
}} while (!isPrinted && ch == 'y');
} finally {
        scanner.close();
}
//Show success message if isPrinted is true
```

Summary

- Stream API is used to traverse and query collections
  - Streams have "lazy" execution
- Streams can be Generic or Primitive
- Types of Operations
- Streams can be collected into a collection
- Streams can operate over Map entries, keys or values.
- Don’t forget to release resources, handle an exception if you can throw it otherwise.
- Some exceptions are checked in compile time, others are not (unchecked)

Java OOP Fundamentals

Questions?

https://softuni.bg/courses/databases-advanced-hibernate

License

- This course (slides, examples, demos, videos, homework, etc.) is licensed under the "Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International" license

- Attribution: this work may contain portions from
  - "Databases" course by Telerik Academy under CC-BY-NC-SA license

Free Trainings @ Software University

- Software University Foundation – softuni.org
- Software University – High-Quality Education, Profession and Job for Software Developers
  - softuni.bg
- Software University @ Facebook
  - facebook.com/SoftwareUniversity
- Software University Forums
  - forum.softuni.bg