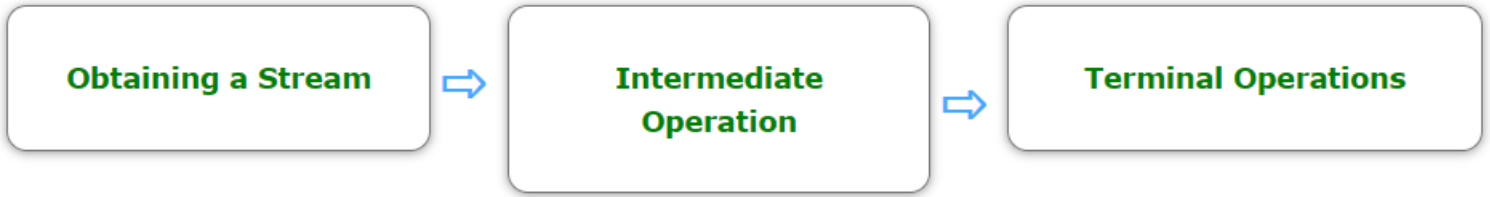
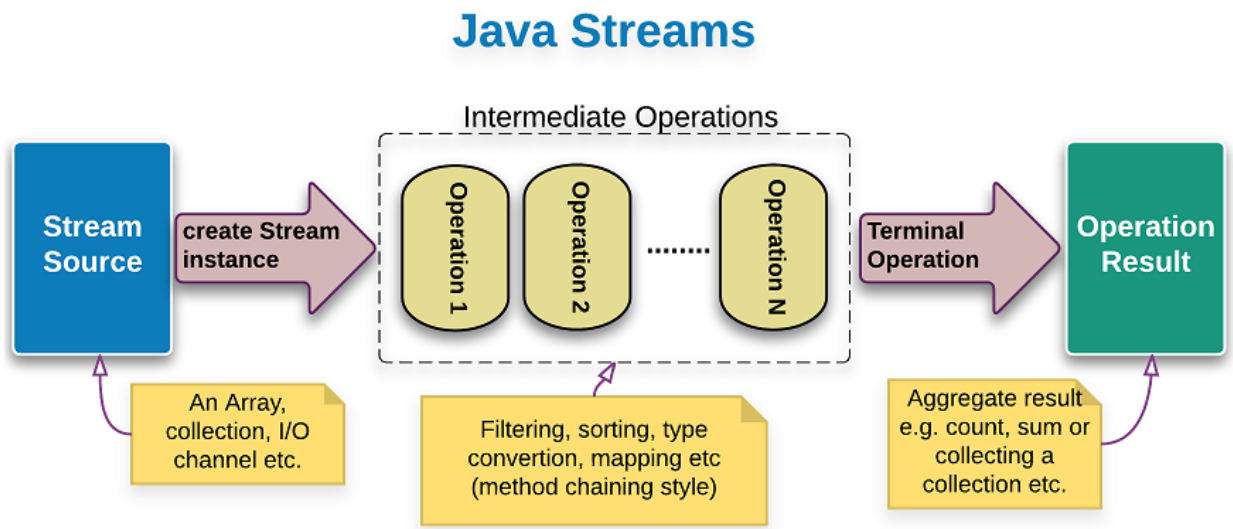
Stream

https://stackify.com/streams-guide-java-8/



Outlines

- Overview
- Java Stream Creation
- Java Stream common operations
- Glance at Java Stream operations Cheat-sheet
- Java Stream useful operations (more)
- Recap (6 Difference Between Intermediate And Terminal Operations In Java)



https://www.logicbig.com/tutorials/core-java-tutorial/java-util-stream/stream-cheat-sheet.html

Overview

- The addition of the Stream was one of the major features added to Java 8. This in-depth tutorial is an introduction to the many functionalities supported by streams, with a focus on simple, practical examples.
  - Re-design the interface
  - Introduce lambda expressions
  - Retrofit the Collection framework.
  - Introduce the Stream API to efficiently handle filter-map-reduce operations in functional programming.
- To understand this material, you need to have a basic, working knowledge of Java 8 (lambda expressions, Optional, method references).
- A stream **does not store data** and, in that sense, is not a data structure. It also **never modifies** the underlying data source.
  - streams are wrappers around a data source, allowing us to operate with that data source and making bulk processing convenient and fast.

https://stackify.com/streams-guide-java-8/

Java Stream Creation

```
private static Employee[] arrayOfEmps = {
  new Employee(1, "Jeff Bezos", 100000.0),
  new Employee(2, "Bill Gates", 200000.0),
  new Employee(3, "Mark Zuckerberg", 300000.0)
};
static List<Employee> empList;
Stream.Builder<Employee> empStreamBuilder;
```

```
static void q1_array_to_Stream() {
  Stream.of(arrayOfEmps);
}

static void q2_list_to_Stream() {
  empList = Arrays.asList(arrayOfEmps);
  empList.stream();
}
```

```
static void
  q3_create_Stream_from_invidual_obj() {

  empStreamBuilder = Stream.builder();

  empStreamBuilder.accept(arrayOfEmps[0]);
  empStreamBuilder.accept(arrayOfEmps[1]);
  empStreamBuilder.accept(arrayOfEmps[2]);

  Stream<Employee> empStream =
                    empStreamBuilder.build();
}
```

https://stackify.com/streams-guide-java-8/

Java Stream common operations (1)

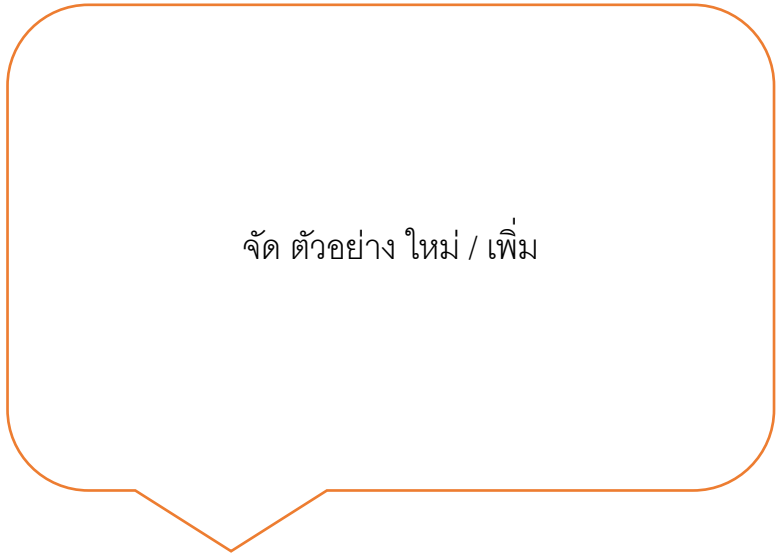
- map() - produces a new stream after applying a **functio**n to each element of the original stream. The new stream could be of different type.
  - peek() – performs the specified operation on each element of the stream and returns a new stream which can be used further
    - https://stackoverflow.com/questions/44370676/java-8-peek-vs-map
- filter() – produces a new stream that contains elements of the original stream that pass a given test (specified by a **Predicate**).
- Flatmap() – A stream can hold complex data structures like *Stream<List<String>>*. In cases like this, *flatMap()* helps us to flatten the data structure to simplify further operations

Java Stream common operations (2)

- forEach - loops over the stream elements, calling the supplied function on each element.
  - forEach() is a terminal operation
- collect() - The strategy for this operation is provided via the Collector interface implementation.
  - If we need to get an array out of the stream, we can simply use toArray()
    - The syntax *Employee[]::new* creates an empty array of *Employee* – which is then filled with elements from the stream.
- findFirst() returns an **Optional** for the first entry in the stream; the Optional can, of course, be empty.

Java Stream common operations (3)

- short-circuiting (intermediate) operations skip(), limit()
- count()



Stream Operations (.forEach(), .map(), .collect())

forEach() loops over the stream elements, calling the supplied function on each element.

```
static public void
q4_whenIncrementSalaryForEachEmployee
_thenApplyNewSalary(double r) {
  //empList from Q2
  empList.stream().forEach(e ->
                e.salaryIncrement(r));
}
```

map() produces a new stream after applying a function to each element of the original stream

```
public static void q5_map_to_getName() {
  empList = Arrays.asList(arrayOfEmps);
  List<String> empNames = empList.stream()
                .map(Employee::getName)
                .collect(Collectors.toList());
  //System.out.println(empNames);
}
```

https://stackify.com/streams-guide-java-8/

Stream Operations (.filter(), .findFirst())

filter() produces a new stream that contains elements of the original stream that pass a given test

```
public static void q6_filter_salary() {
  Employee[] ans =
        Arrays.asList(arrayOfEmps)
              .stream()
              .filter(e ->
                e.getSalary() > 200000)
              .toArray(Employee[]::new);
  println(Arrays.toString(ans));
}
```

https://stackify.com/streams-guide-java-8/

findFirst() returns an Optional for the first entry in the stream

```
public static void q7_findFirst() {
  Optional<Employee> opt = empList
                .stream()
                .filter(e ->
                  e.getSalary() > 400_000)
                .findFirst();
  opt.ifPresent(System.out::println); // none

empList.stream()
        .filter(e -> e.getSalary() > 100_000)
        .findFirst()
        .ifPresent(System.out::println);
}
```

Stream Operations (.flatmap())

```
public static void q8_flatMap() {
  List<String> teamA
         = Arrays.asList("yindee");
  List<String> teamB
         = Arrays.asList("preeda", "pramote");
  List<String> teamC
         = Arrays.asList("sukha");

  List<List<String>> allTeams = new ArrayList<>();
    allTeams.add(teamA);
    allTeams.add(teamB);
    allTeams.add(teamC);

  List<String> allPlayers
           = allTeams.stream()
                  .flatMap(team -> team.stream())
                  .collect(Collectors.toList());
  //System.out.println(allPlayers);
```

flatMap() helps us to flatten the data structure to simplify further operations:

```
  List<List<Employee>> jeffBillMark
        = Arrays.asList(
             Arrays.asList(arrayOfEmps[0]),
             Arrays.asList(arrayOfEmps[1]),
             Arrays.asList(arrayOfEmps[2]) );
  // https://stackify.com/streams-guide-java-8/
  List<Double> ans = jeffBillMark
                  .stream()
                  .flatMap(Collection::stream)
                  .map(Employee::getSalary)
                  .collect(Collectors.toList());
  //System.out.println(ans);
}
```

Stream Operations (.peek(), .count())

peek() method exists mainly to support debugging, where you want to see the elements as they flow past a certain point in a pipeline.

***peek()* is an intermediate operation**:

```
static void q9_peek() {
  List<Employee> newSalary = empList.stream()
                                    .peek(e -> e.salaryIncrement(10))
                                    .peek(System.out::println)
                                    .collect(Collectors.toList());
  // System.out.println(newSalary);
}
```

```
static void q10_count() {
  Long empCount = empList.stream()
                         .filter(e -> e.getSalary() > 200000)
                         .count();
  System.out.println(empCount); // 1
}
```

https://stackify.com/streams-guide-java-8/

Stream Operations (.skip(), .limit())

Some operations are deemed **short-circuiting operations**. Short-circuiting operations allow computations on infinite streams to complete in finite time

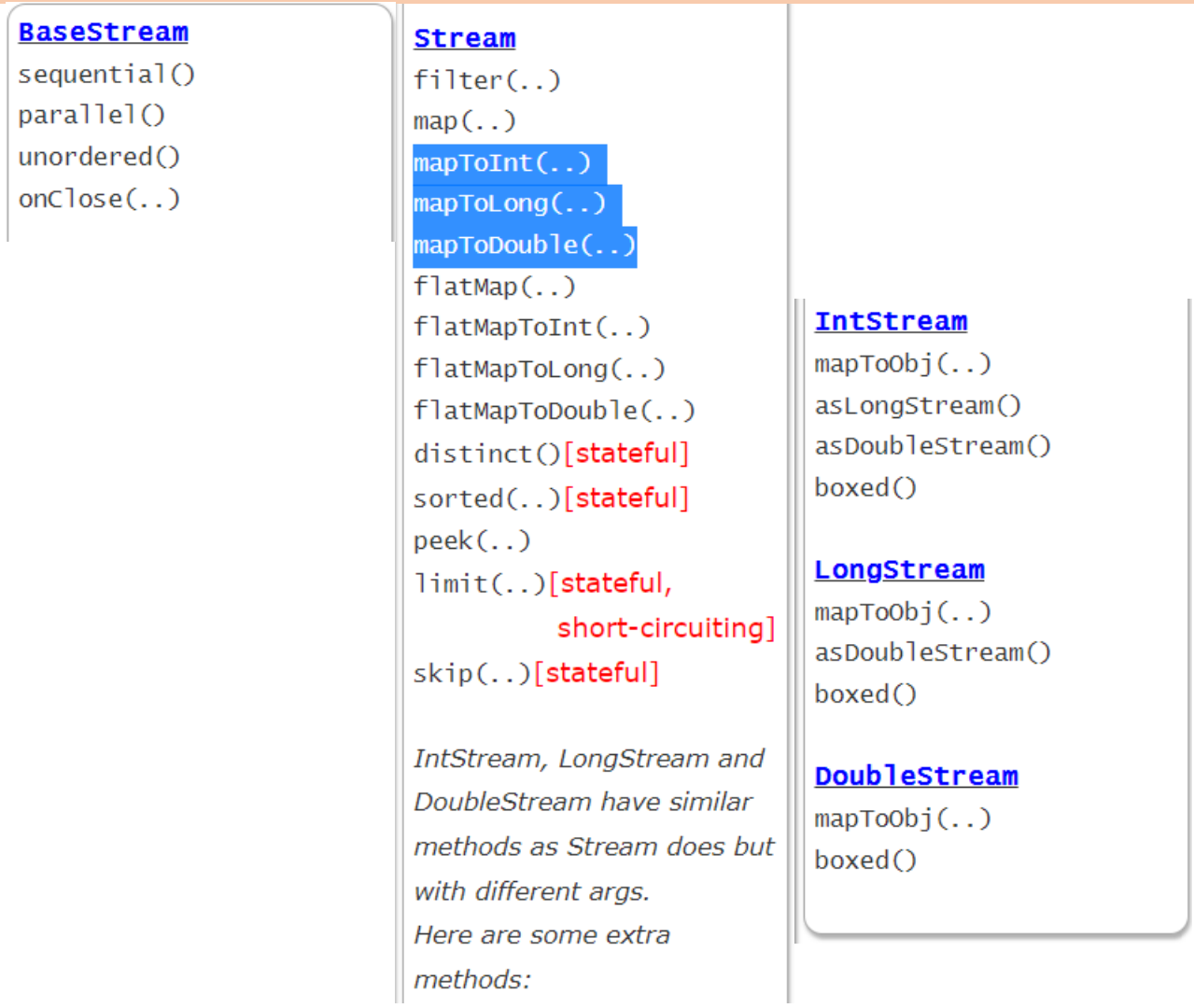
```
static void q11_limit_and_skip() {
  List<Employee> theRest = empList.stream()
                  .skip(1)
                  .collect(Collectors.toList());
  // System.out.println(theRest);

  List<Employee> head2 = empList.stream()
                  .limit(2)
                  .collect(Collectors.toList());
  // System.out.println(head2);

  // for infinite data generator see Stream.iterate()
  // or Stream.generate()
}
```

Glance at Java Stream operations Cheat-sheet

- Stream source

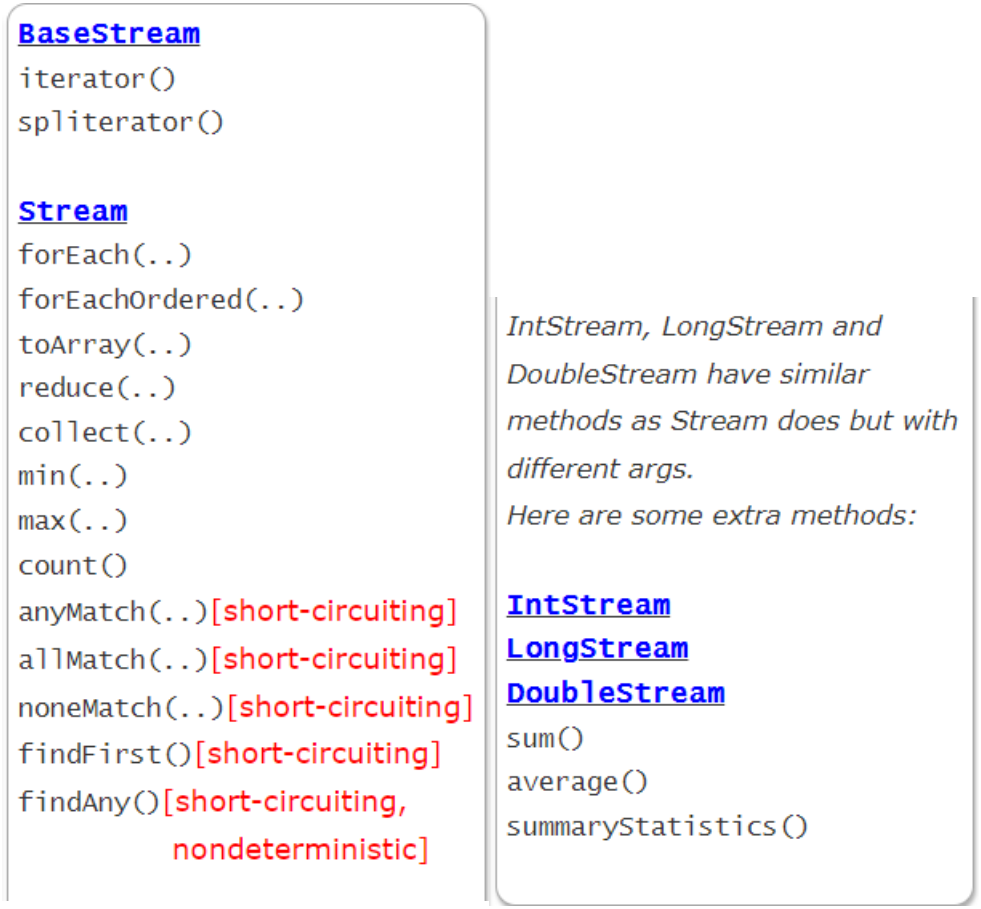
Glance at Java Stream operations Cheat-sheet

- Intermediate Operation



Glance at Java Stream operations Cheat-sheet

- Terminal Operation



Java Stream useful operations (more)

- Comparison Based Stream Operations
  - Use Comparator
  - Sort()
  - min() and max()
- Java Stream Specializations
  - IntStream, LongStream, and DoubleStream
  - mapToInt(), mapToDouble(), mapToLong()
- distinct(), allMatch(), sum(), average(), range()
- partitioningBy()
- groupBy()
- Reduce()
- parallel()

Comparator, sort(),

```
static void q12_sortByName() {
        List<Employee> byName;
        byName = empList.stream()
                .sorted(Comparator.comparing(Employee::getName))
                .collect(Collectors.toList());
        // System.out.println(byName);

        byName = empList.stream()
                .sorted((e1, e2) -> e1.getName().compareTo(e2.getName()))
                .collect(Collectors.toList());
        // System.out.println(byName);
}
```

Comparator, mapToDouble(), max()

```
static void q13_mapToDouble() {
  Double maxSalary;
  maxSalary
    = empList.stream()
        .mapToDouble(e -> e.getSalary())
        .max(/* no-comparator-required */)
        .getAsDouble();
  // System.out.println(maxSalary);

  Optional<Double> optSalary;
  optSalary
    = empList.stream()
              .map(e -> e.getSalary())
              .max(Double::compareTo);
  maxSalary = optSalary.get();
  // System.out.println(maxSalary);
```

```
Optional<Employee> optEmp;
optEmp = empList.stream()
            .max(Comparator.comparing(
                        Employee::getSalary) );
maxSalary = optEmp.get().getSalary();
// System.out.println(maxSalary);

optEmp = empList.stream()
             .max( (e1, e2) ->
 Double.compare(e1.getSalary(), e2.getSalary()));
maxSalary = optEmp.get().getSalary();
// System.out.println(maxSalary);

// optEmp = empList.stream().max(
// (e1, e2) ->
//      (int)(e1.getSalary() - e2.getSalary()));
// maxSalary = optEmp.get().getSalary();
// wrong logic 0.9 > 0.3 but (int)(0.9 - 0.3) is 0
}
```

groupBy()

```
static void q14_groupBy() {
  Character[] forRetrieval = { 'M', 'J', 'B' };

  Map<Character, List<Employee>> mapOfFirstCharacter;
  mapOfFirstCharacter = empList.stream()
          .collect(
                  Collectors.groupingBy(e -> Character.valueOf(e.getName().charAt(0))));
       for (Character ini : forRetrieval) {
           String empName = mapOfFirstCharacter.get(ini).get(0).getName();
           // System.out.println(empName);
       }
    }
```

reduce()

reduce

The most common form of *reduce()* is:

```
T reduce(T identity, BinaryOperator<T> accumulator)
```

where *identity* is the starting value and *accumulator* is the binary operation we repeated apply.

```
static void q15_reduce() {
    LineItem[] arrItems = { new LineItem("Mocha", 10, 10.2),
            new LineItem("cappuccino", 5, 16.2), new LineItem("Latte", 20, 5.1),
            new LineItem("Espresso", 8, 11.39) };
    List<LineItem> orders = Arrays.asList(arrItems);

    double total = orders.stream().map(LineItem::quantityTimesPrice)
            .reduce(Double::sum).get();
    // System.out.println(total);
    double min = orders.stream().map(item -> item.quantityTimesPrice())
            .reduce(Double.MAX_VALUE, (v1, v2) -> v1 < v2 ? v1 : v2);
    // System.out.println(min);
    String concatName = orders.stream()
            .map(LineItem::getName)
            .reduce("", String::concat);
    // System.out.println(concatName);
    int productOfNums = List.of(2, 5, 7).stream().reduce(1, (e1, e2) -> e1 * e2);
    // System.out.println(productOfNums);
}
```

Differences Between Intermediate And Terminal Operations In Java

1. Return type

- Intermediate operations return a stream itself while Terminal operations produce either a value or a side-effect.

Intermediate operations return a stream itself. Example is:

```
Stream<T> distinct()
```

Terminal operations produce either a value or a side-effect.

```
Optional<T> findAny()
```

2,3 Number of operations

- An intermediate operation returns a new stream , they can be chained.
- A stream pipeline must have one and only one terminal operation.

4,5 Intermediate operations are not executed until a terminal operation is performed.

- laziness also allows avoiding examining all the data when it is not necessary.

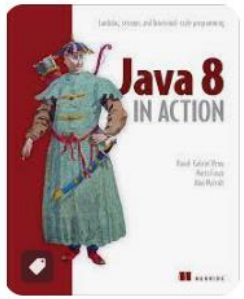
6. Short-Circuiting Operation

- one intermediate operation exhibits Short-Circuiting: limit().
- There are five terminal operations, which can exhibit Short-Circuiting operation: findFirst(), allMatch(), anyMatch(), findAny(), noneMatch().

https://javahungry.blogspot.com/2022/07/intermediate-vs-terminal-operations.html

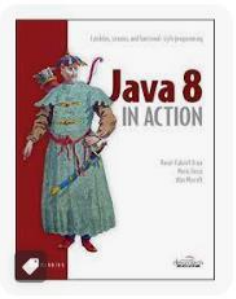
Recommended Reading

Manning Publications
Java 8 in Action



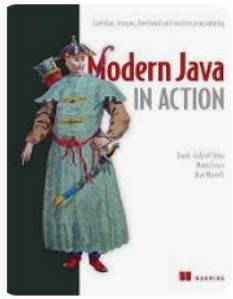
Amazon.c... · In stock
Java 8 in Action: La...

Amazon.com
Java 8 in Action: Lambda...



Amazon.in · In stock
Java 8 in Action: La...

Amazon.ae
Java 8 in Action: Lambdas, Streams...



Amazon
Modern Java in Acti...

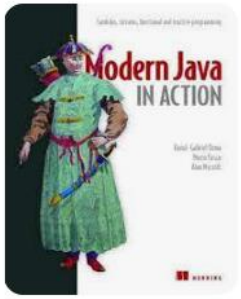
Manning Publications
Java 8 in Action

Amazon
Modern Java in Acti...

Internet Archive
Raoul-Gabriel Urma ...

PDF Room
Java 8 in Action: Lambdas, ...

Amazon.c... · In stock
Java 8 in Action: La...

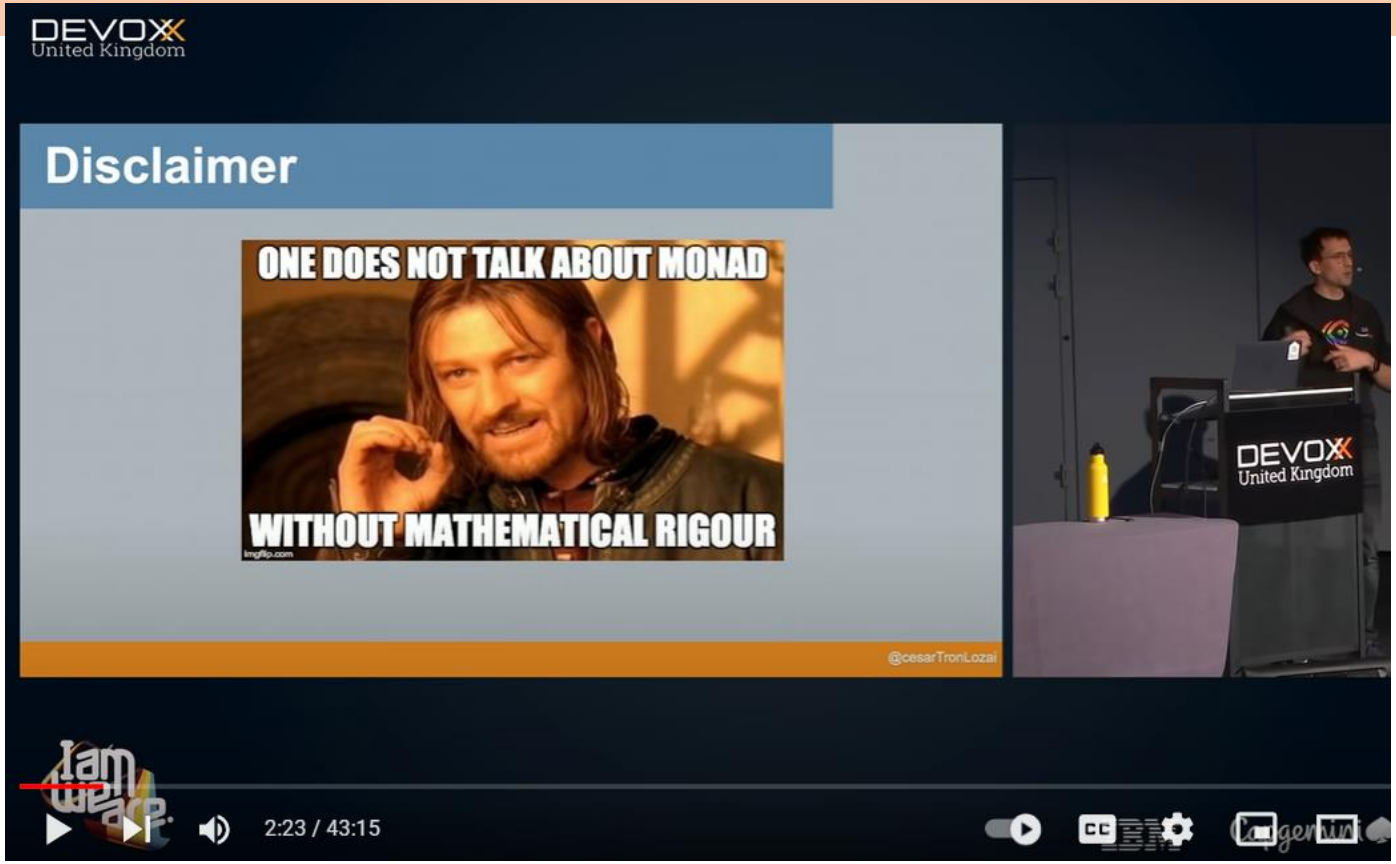


Paper Plus
Modern Java in Acti...

Audible
Java 8 in Action by Raoul-Gabriel Urma ...

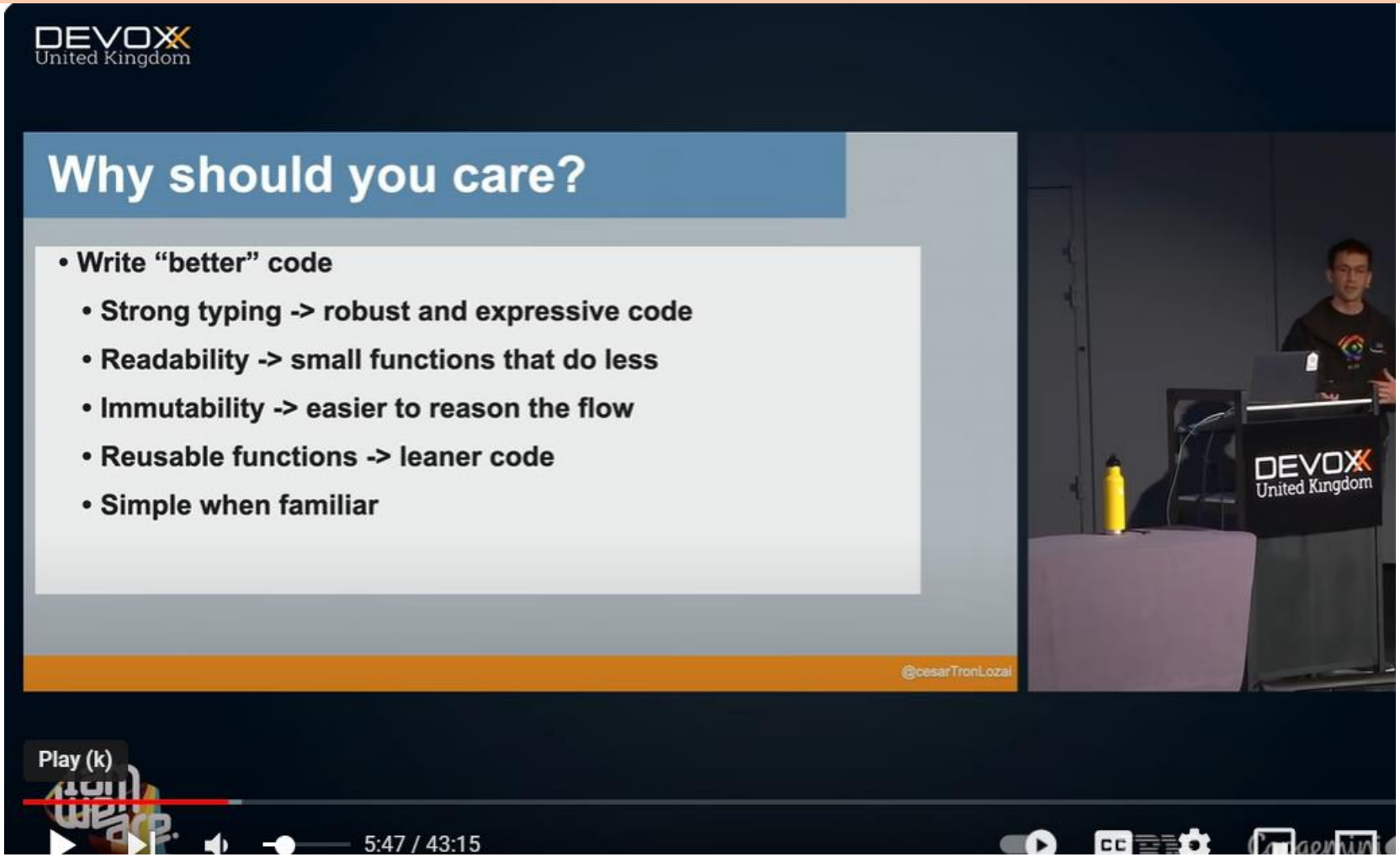
국내도서 - 교보문고
자바 8 인 액션 | 라...

Appendix : Functional Programming Style

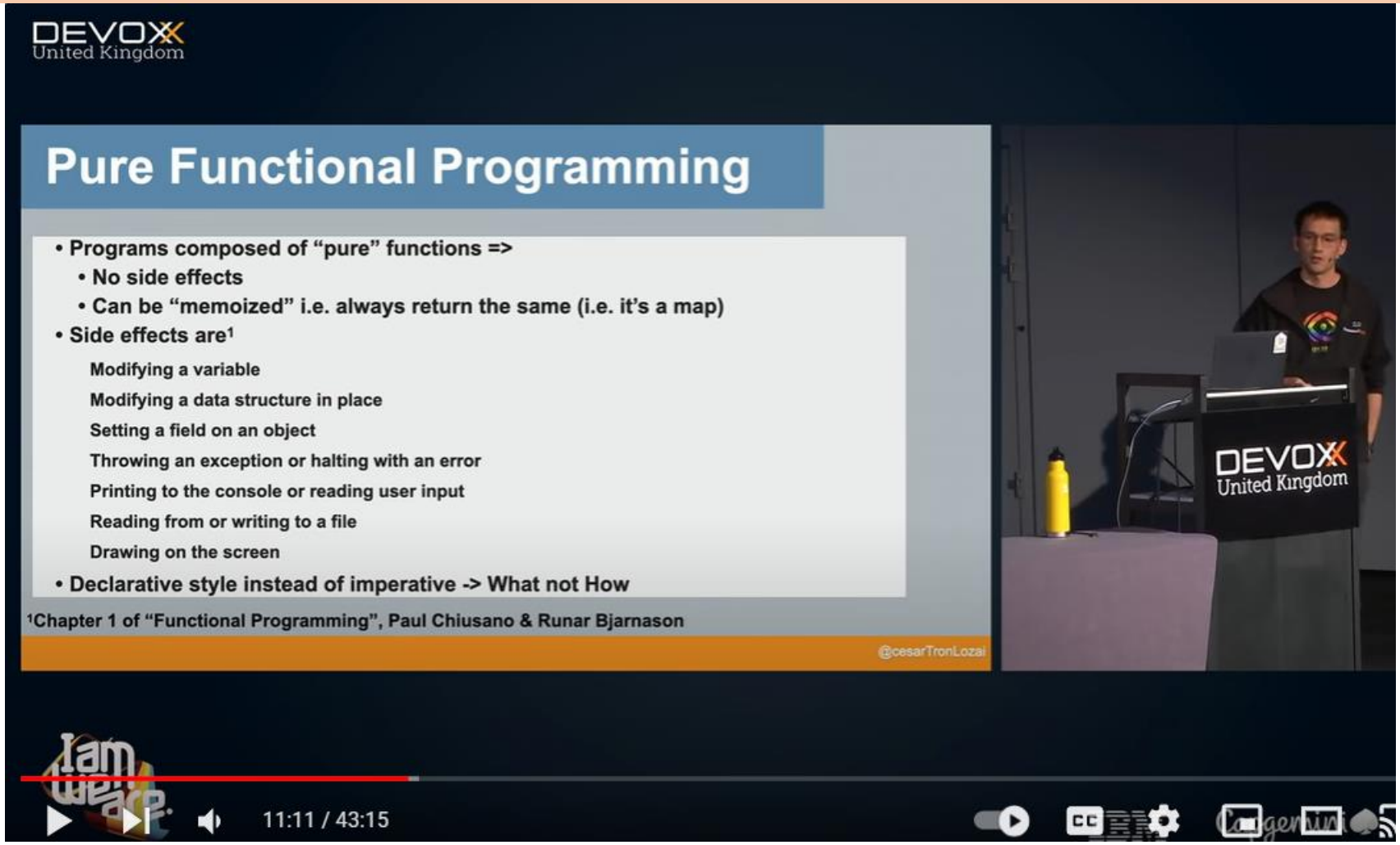


No Nonsense Monad & Functor - The foundation of Functional Programming by César Tron-Lozai
https://www.youtube.com/watch?v=e6tWJD5q8uw&t=1s

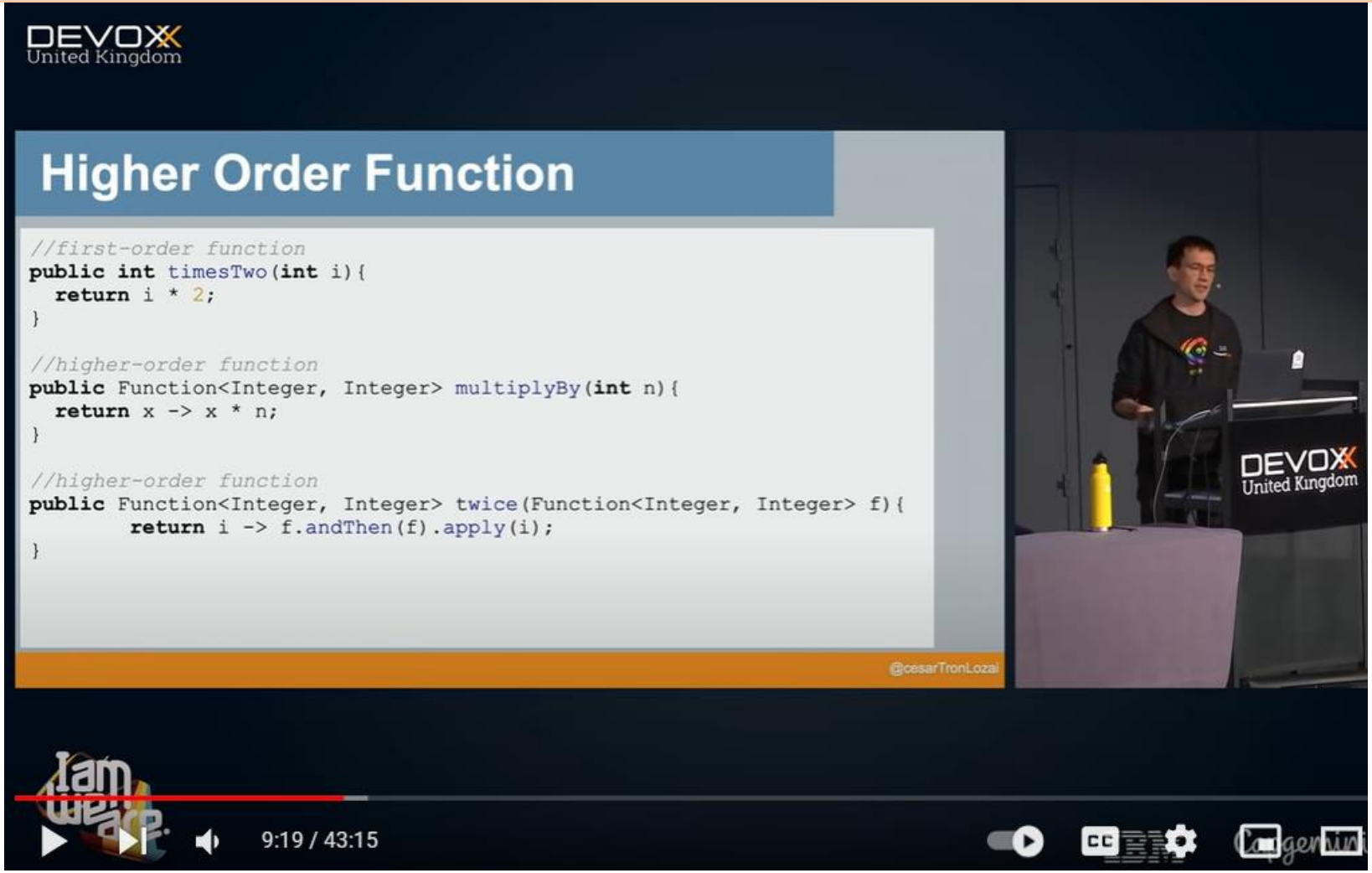
Appendix : Functional Programming Style



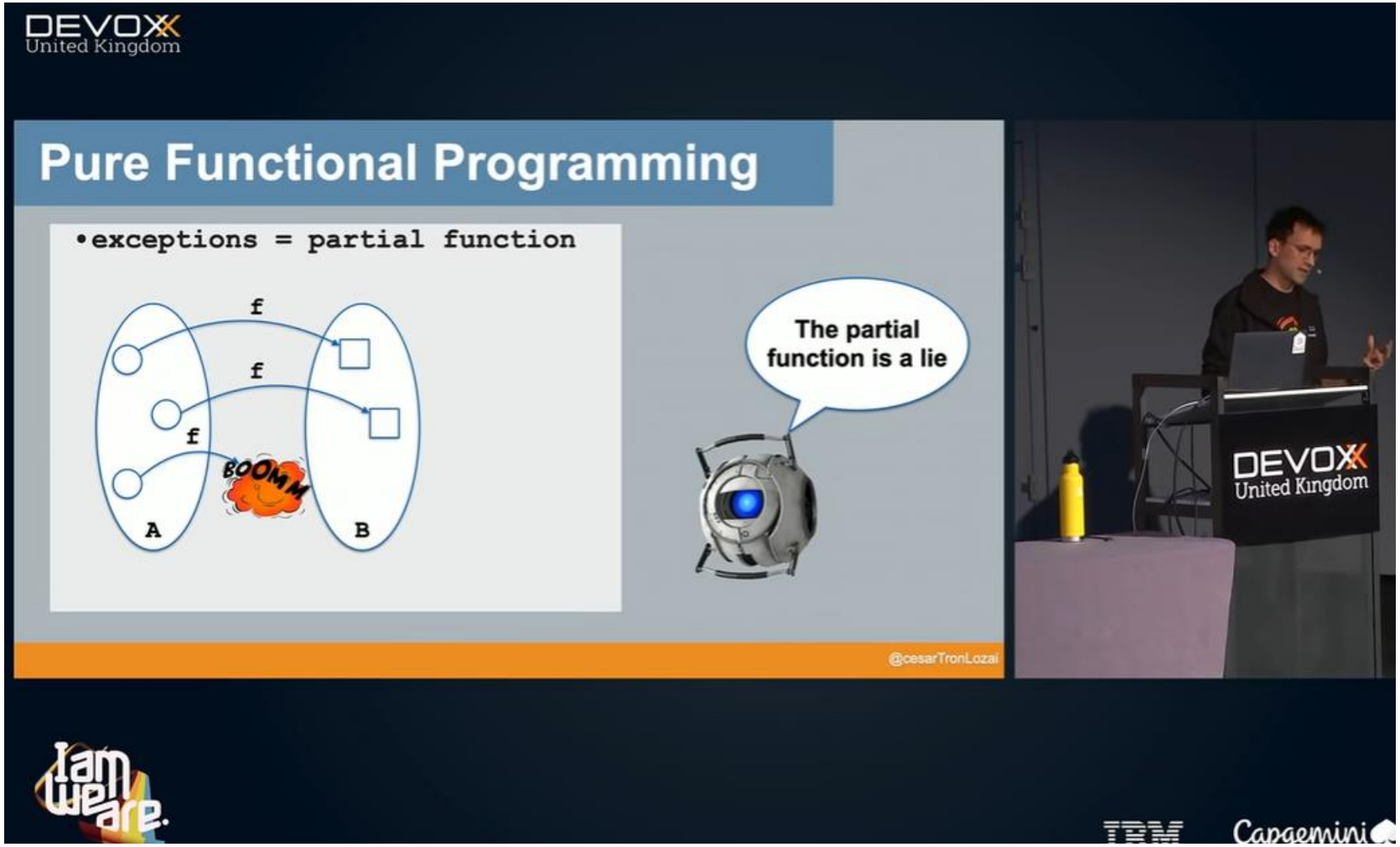
Appendix : Functional Programming Style



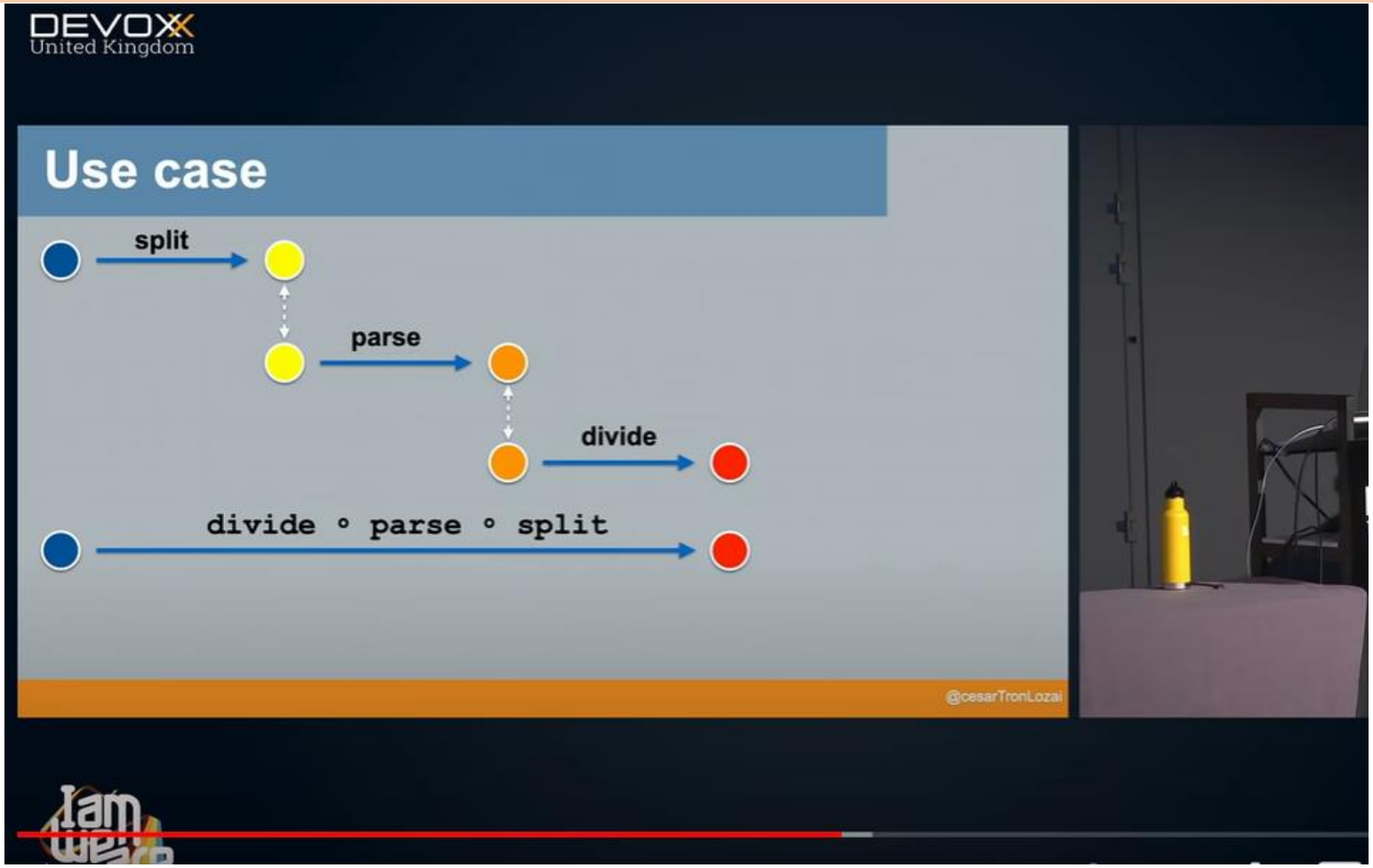
Appendix : Functional Programming Style



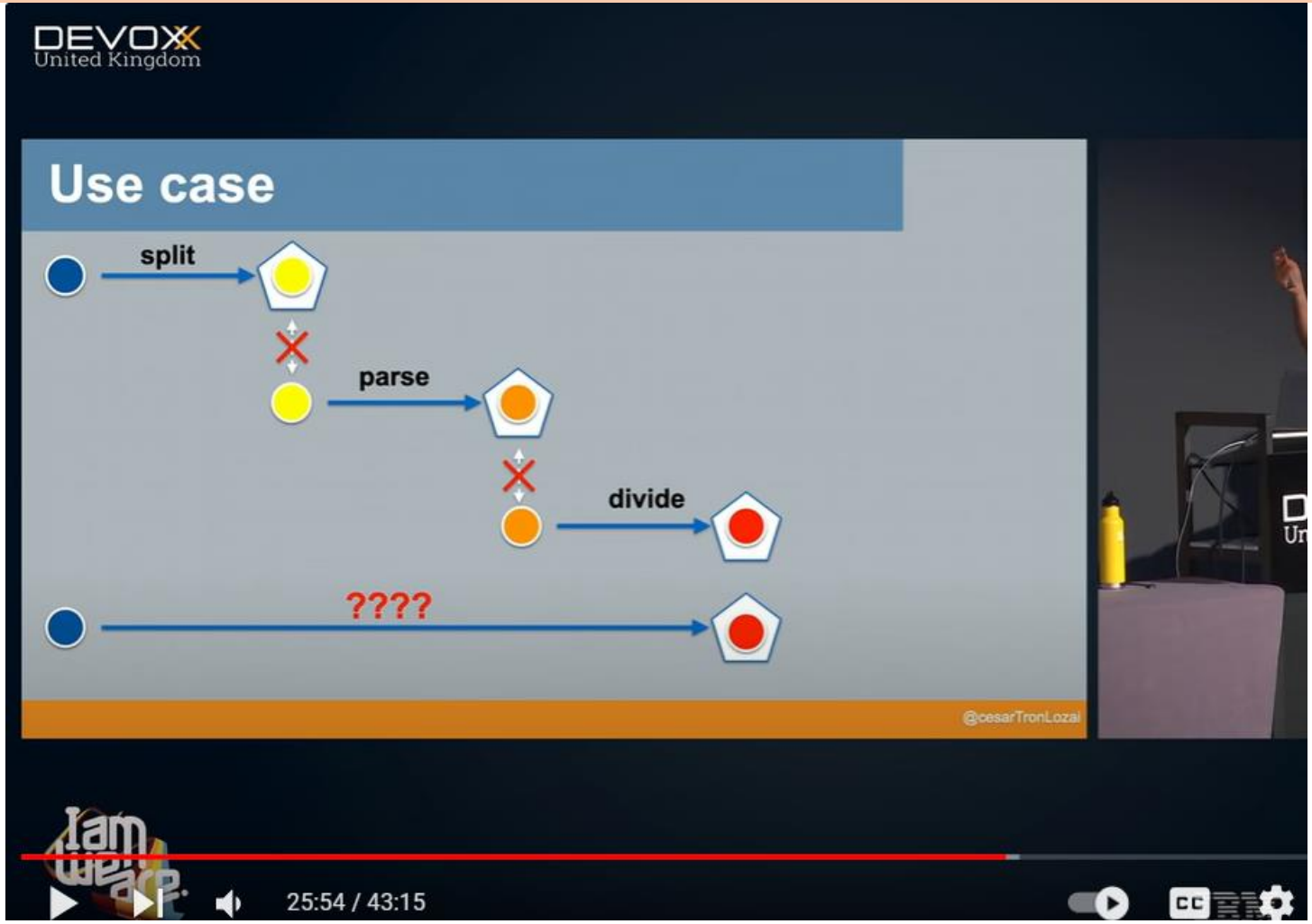
Appendix : Functional Programming Style



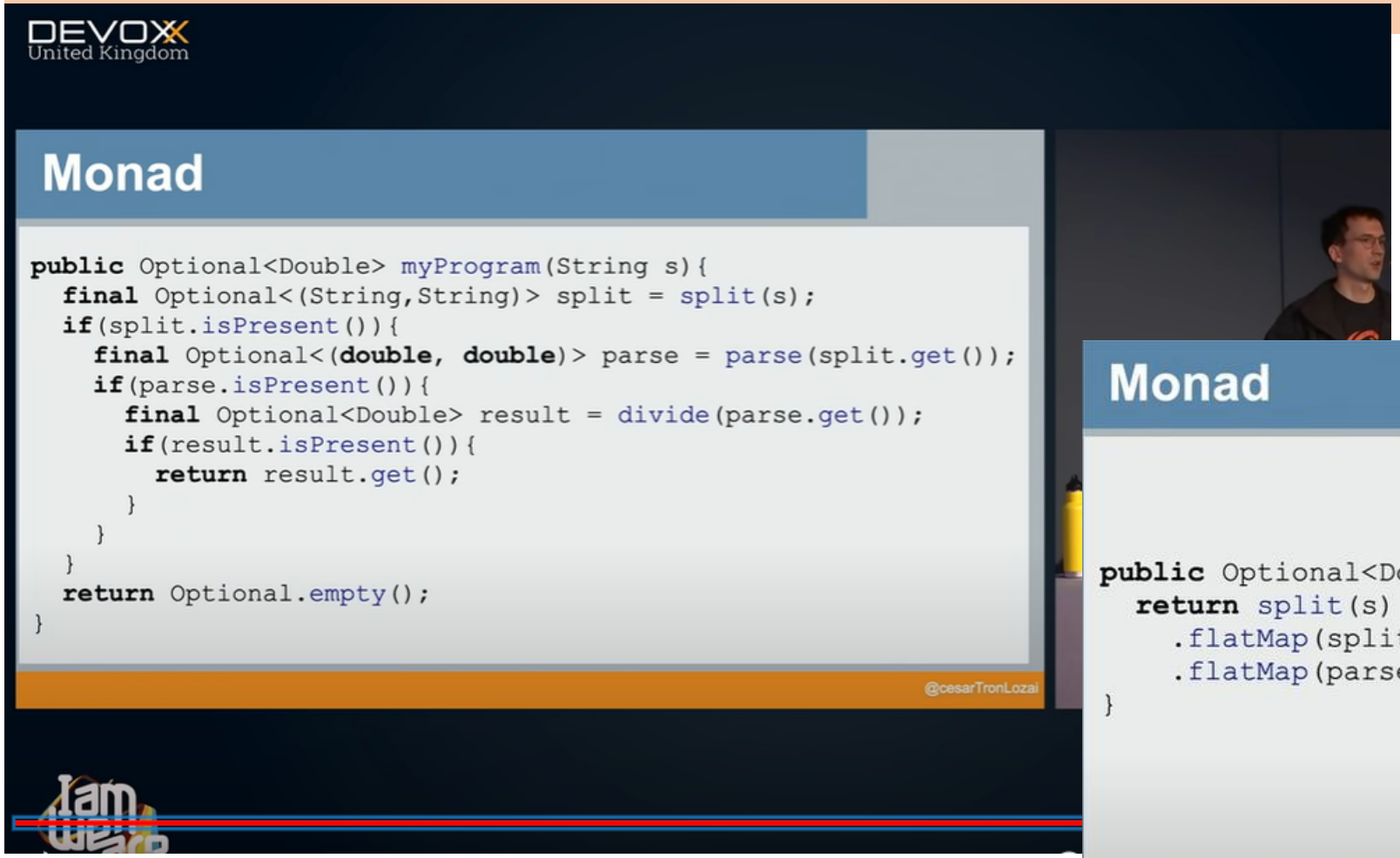
Appendix : Functional Programming Style

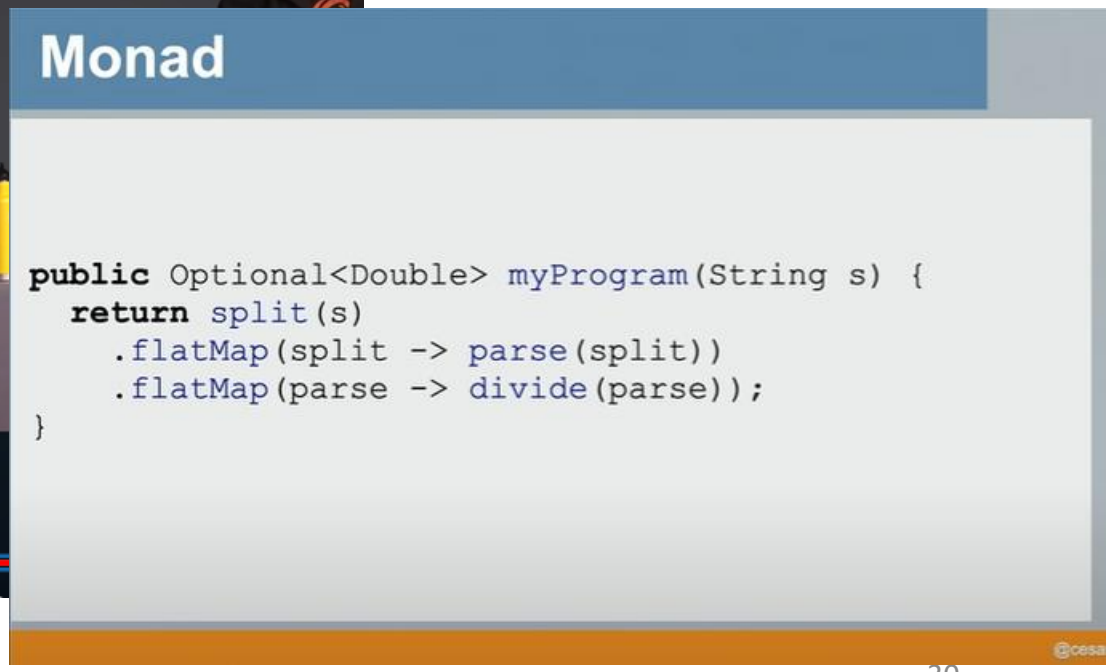


Appendix : Functional Programming Style



Appendix : Functional Programming Style



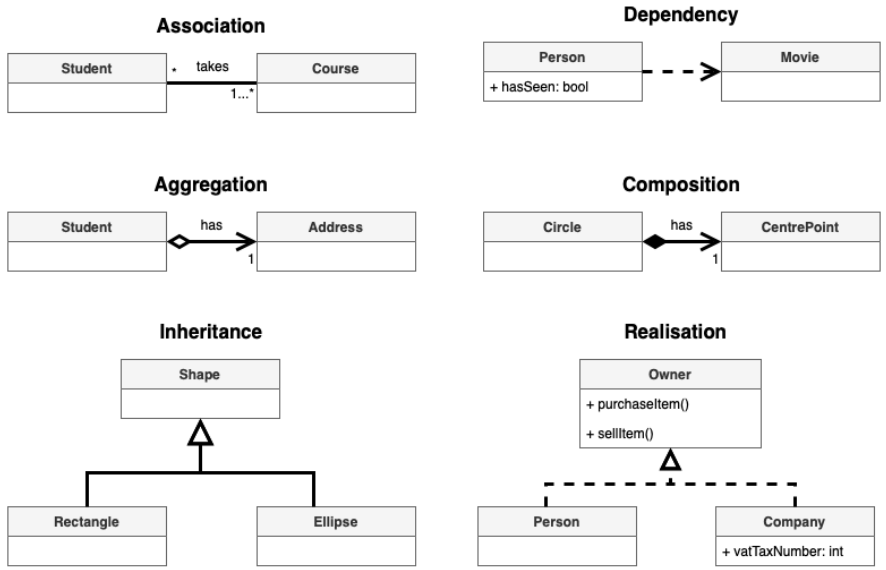


Optional in Java

05506004
https://javarevisited.blogspot.com/2017/04/10-examples-of-optional-in-java-8.html

Outline

- What Actually is Optional
- Create Opitonal Object
- Avoiding NullPointerException
- .map (and .filter())



https://www.diagrams.net/blog/uml-class-diagrams

What Actually is Optional

1. What actually is Optional?

`Optional` is simply a Java class located in the `java.util` package and available within the Java Development Kit (JDK). That means we can use it in anywhere in our Java code, simply by creating an instance of that class.

But before we do that, it's helpful to understand that `Optional` is simply a container for an object. And just like real-life containers, it can have one of 2 states:

1. contains an item (also known as *present*)
2. doesn't contain an item (also known as *empty*)

https://tomgregory.com/java-optional/

Take my glasses case, for example. It either contains my glasses or doesn't contain my glasses.

My glasses case is a kind of container, just like `Optional`.

Create Opitonal Object

```
Person p = new Person();
Optional<Person> op = Optional.of(p);
```

```
Optional<Person> p = Optional.empty();
```

```
Optional<Person> op = Optional.ofNullable(p);
```

Avoiding NullPointerException

```
Person p = new Person("Robin", new Address(block, city, state, country);
Address a = p.getAddress();

if(a != null){
 System.out.println(p);
}
```

Now, in Java 8 you can completely avoid this check by using the `isPresent()` method of the Optional class, which allows you to execute code if a value is printed and the code will not execute if no value is there, as shown in the following example:

```
Optional<Address> op = p.getAddress();
op.isPresent(System.out::println);
```

```
if(!op.isPresent()){
 System.out.println(p);
}
```

This is not recommended because it is similar to classical check and not the right way to use Optional in Java SE 8. You can further read Java 8 in Action to learn more about how to use Optional in Java SE 8.

```
public class PersonRevisitedBlog {
  private String name;
  private Optional<City> fromCity;

  public PersonRevisitedBlog(String n,
                             Optional<City> city) {
    try {
        if (city.isPresent()) {
            fromCity = city;
        } else {
            fromCity = Optional.empty();
        }
    } catch (Exception e) {
        fromCity = Optional.empty();
        // fromCity = Optional.ofNullable(null);
    }
    // if (city != null)
    //   fromCity = city;
    // else
    //   fromCity = Optional.empty();
    }
                                          }
```

```
public class City {
  public static final
         City EMPTY = new City("");
  private String cName;

  public City(String n) {
      cName = n;
  }

  public String getCityName() {
      return cName;
  }

  @Override
  public String toString() {
     return cName;
  }
}
```

```
public Optional<City> getCity() {
    return fromCity;
}
```

```java
static void q1() {
  City bkk = new City("Bangkok");
  City city;

  PersonRevisitedBlog yindee =
       new PersonRevisitedBlog("Yindee", Optional.of(bkk));

  city = yindee.getCity().orElse(City.EMPTY);
  println("From yindee : " + city.getCityName());

  Optional<City> optCity1 = yindee.getCity();
  System.out.println(optCity1);
  optCity1.ifPresent(System.out::println);

  println("--------");
  PersonRevisitedBlog preeda =
       new PersonRevisitedBlog("Preeda", Optional.empty());

  // Optional<City> optCity2 = preeda.getCity();
  optCity1 = preeda.getCity();
  if (optCity1.isPresent()) {
    city = optCity1.get();
    println("From preeda : " + city.getCityName());
  }
  System.out.println("--------");
  PersonRevisitedBlog pramote = new
PersonRevisitedBlog("Pramote", null);
  // String cityName
    = city == null? "" : city.getCityName();
  city = pramote.getCity()
           .orElse(City.EMPTY);
  System.out.println("From pramote : "
                     + city.getCityName());
}
```

Optional.of(new City())

```
From yindee : Bangkok
Optional[Bangkok]
Bangkok
--------
--------
From pramote :
```

.map (and .filter())

How to use map method with Optional in Java 8

The `map()` method of the Optional class is similar to the `map()` function of Stream class, hence if you have used it before, you can use it in the same way with Optional as well. As you might know, `map()` is used to transform the object i.e. it can take a String and return an Integer. It applies a function to the value contained in the `Optional` object to transform it.

For example, let's say you want to first check if a person is not null and then want to extract the address, this is the way you would write code prior to Java 8

```
if(person != null){
  Address home = person.getAddress();
}
```

You can rewrite this check-and-extract pattern in Java 8 using Optional's map() method as shown in the following example:

```
Optional<Address> = person.map(person::getAddress);
```

Here we are using the method reference to extract the Address from the Person object, but you can also use a lambda expression in place of method reference if wish. If you want to learn more about when you can use a lambda expression and when method reference is appropriate, you should a good book on Java 8. You can find some recommended Java 8 books here.

How to use filter method with Optional in Java 8

Similar to the Stream class, Optional also provides a filter() method to select or weed out unwanted values. For example, if you want to print all persons living in NewYork, you can write following code using the filter method of the Optional class:

```
Optional<Address> home = person.getAddress();
home.filter(address -> "NewYork".equals(address.getCity())
    .ifPresent(() -> System.out.println("Live in NewYork"));
```

This code is safe because it will not throw any `NullPointerException`. This will print "Live in NewYork" if a person has address and city are equal to "NewYork". If the person doesn't have any address then nothing would be printed.

Just compare this to the old style of writing safe code prior to Java 8 e.g. in JDK 6 or 7:

```
Address home = person.getAddress();
if(home != null && "NewYork".equals(home.getCity()){
  System.out.println("NewYorkers");
}
```

The difference may not be significant in this case but as the chain of objects increases e.g. `person.getAddress.getCity().getStreet().getBlock()`, the first one will be more readable than the second one which will have to perform nested null checks to be safe. You can check out these Java 8 books, courses, and tutorials to learn more about how to write functional code using Optional in Java.

trivial

How to use flatMap method of Optional in Java 8

The `flatMap()` method of Optional is another useful method that behaves similarly to the Stream.flatMap() method and can be used to replace unsafe cascading of code to a safe version. You might have seen this kind of code a lot while dealing with hierarchical objects in Java, particularly while extracting values from objects created out of XML files.

```
String unit = person.getAddress().getCity().getStreet().getUnit();
```

This is very unsafe because any of the objects in the chain can be null and if you check null for every object then the code will get cluttered and you will lose readability. Thankfully, you can use the `flatMap()` method of the Optional class to make it safe and still keep it readable as shown in the following example:

```
String unit= person.flatMap(Person::getAddress)
                   .flatMap(Address::getCity)
                   .flatmap(City::getStreet)
                   .map(Street::getUnit)
                   .orElse("UNKNOWN");
```

Important Points

- 1) The Optional class is a container object which may or may not contain a non-null value. That's why it is named Optional.
- 2) If a non-value is available then Optional.isPresent() method will return true and get() method of Optional class will return that value.
- 3) The Optional class also provides methods to deal with the absence of value e.g. you can call Optional.orElse() to return a default value if a value is not present.
- 4) The java.util.Optional class is a value-based class and use of identity sensitive operations e.g. using the == operator, calling identityHashCode() and synchronization should be avoided on an Optional object.
- 5) You can also use the orElseThrow() method to throw an exception if a value is not present.

Important Points

- 6) There are multiple ways to create Optional in Java 8. You can create Optional using the static factory method Optional.of(non-null-value) which takes a non-null value and wrap it with Optional. It will throw NPE if the value is null. Similarly, the Optional.isEmpty() method return an empty instance of Optional class in Java.
- 7) The biggest benefit of using Optional is that it improves the readability and conveys information which fields are optional.
  Earlier it wasn't possible to convey client which fields are optional and which are always available, but now, if a getter method returns Optional then the client knows that value may or may not be present.
- 8) Similar to java.util.stream.Stream class, Optional also provides filter(), map(), and flatMap() method to write safe code in Java 8 functional style. The method behaves similarly as they do in Stream class, so if you have used them before, you can use them in the same way with the Optional class as well.
- 9) You can use the **map()** method to transform the value contained in the Optional object and flatMap() for both transformations and flattening which would be required when you are doing the transformation in a chain as shown in our Optional + flatMap example above.
- 10) You can also use the **filter()** method to weed out any unwanted value from the Optional object and only action if Optional contains something which interests you.