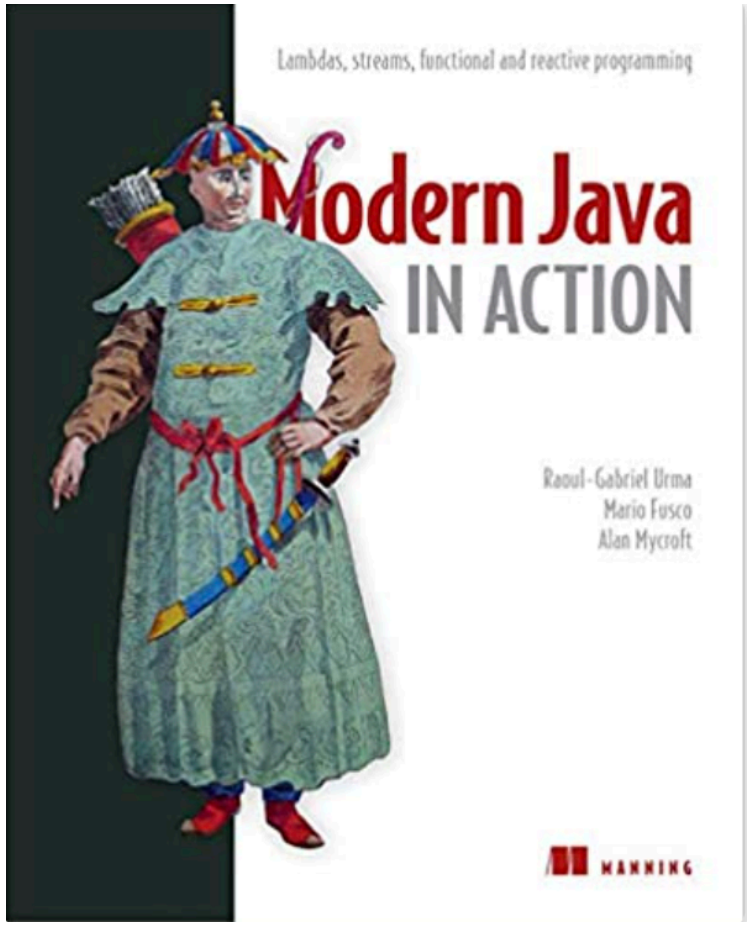


Modern Java in Action: Lambdas, streams, functional and reactive programming
by Raoul-Gabriel Urma, Mario Fusco, Alan Mycroft
Manning, 2nd edition, 2018. Chapters 2, 3, 4, 5.

Create a program useful for adding and filtering products inside a warehouse. Each product has a color, weight, price, description and producer.

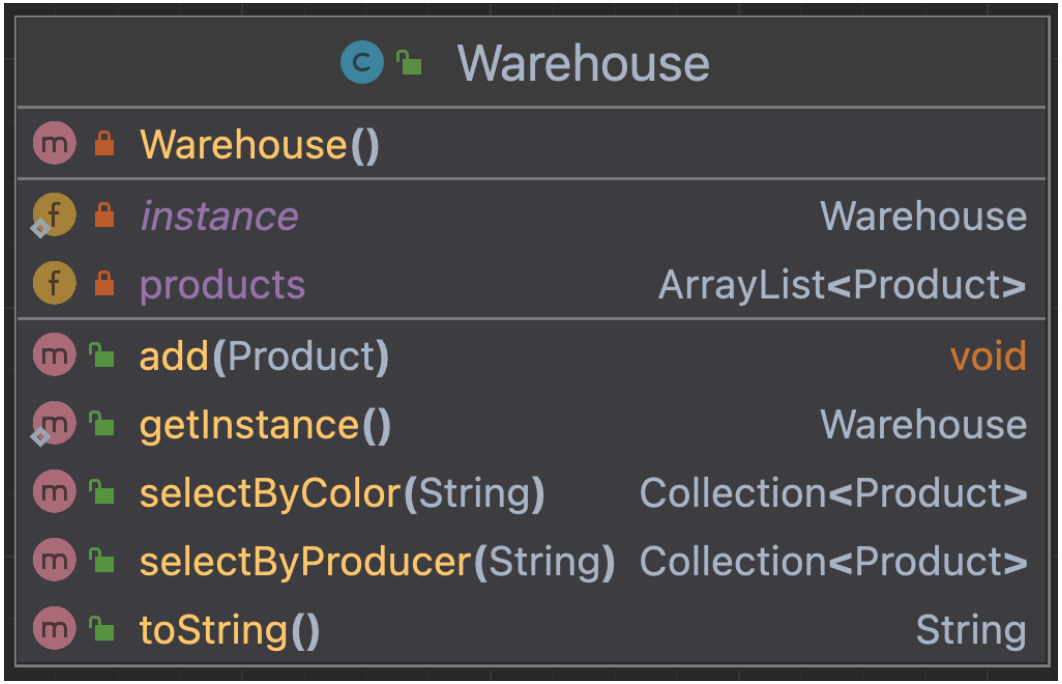
A Warehouse is a class that cannot be instantiated more than once. The single instance of Warehouse provides the following services:

- a method for adding a product into a collection (ArrayList), the collection being an attribute of class Warehouse. If the product is already stored in the warehouse, the client of this class is informed about this situation. The method returns void.
- a method for printing the characteristics of the available products

We need to select from the warehouse (a collection is returned by each service):

- all the products with a given color
- all the products with a given weight
- all the products with a given price
- all the products with a given description
- all the products with a given producer
- all the products with a given color and weight
- all the products with a given color, weight and price
- all the products with a given producer and below a price
- all the products with a given producer and below a price
- all the products with a given producer and over a price
- … all the products with any possible combination among the available characteristics!!!

An example (from OOP)…


Warehouse
Warehouse()
instance
Warehouse
products
ArrayList<Product>
add(Product)
void
getInstance()
Warehouse
selectByColor(String)
Collection<Product>
selectByProducer(String)
Collection<Product>
toString()
String

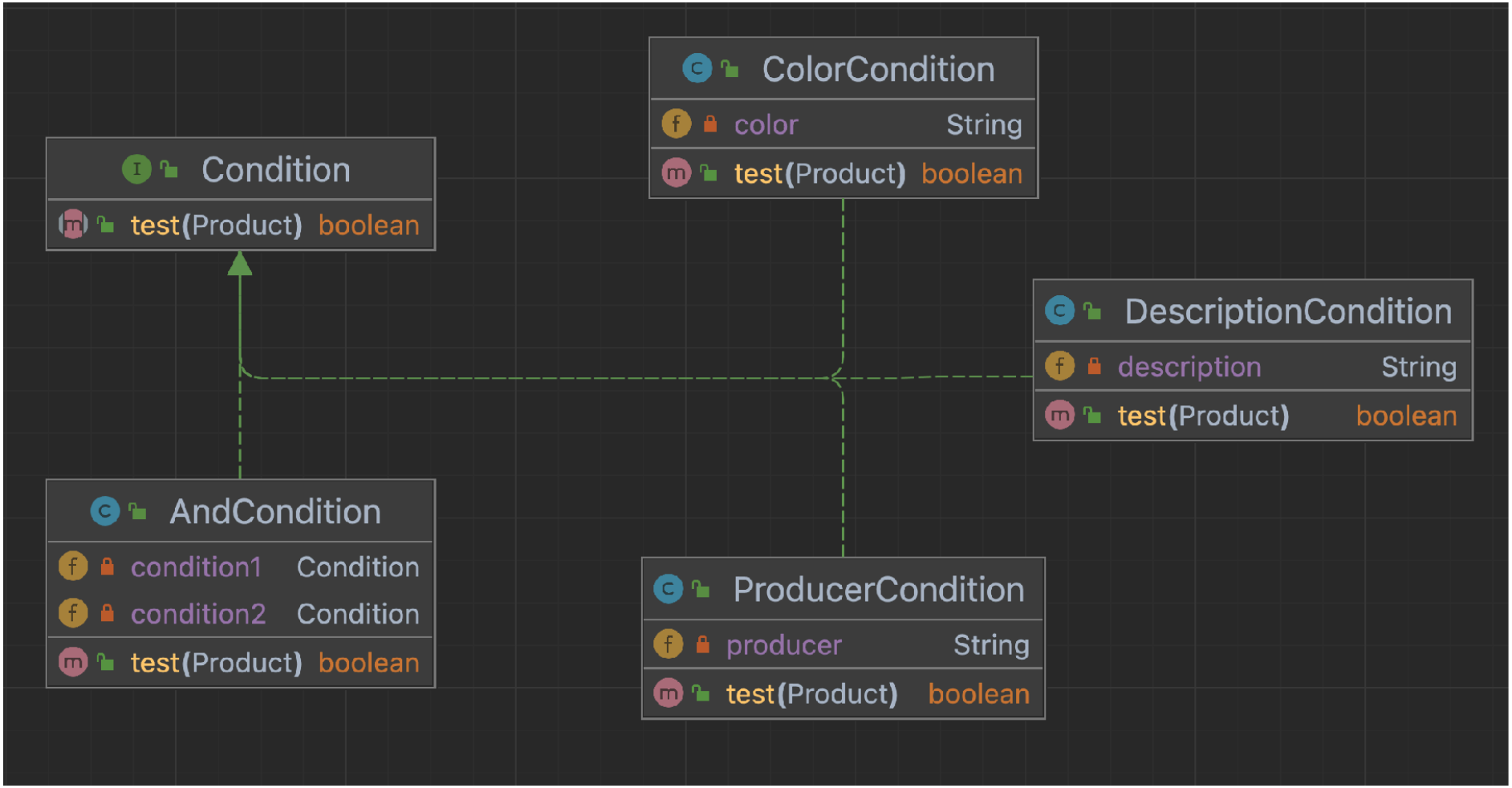
```java
public Collection<Product> selectByColor(String color) {
    Collection<Product> selectedElements = new ArrayList<Product>();

    for(Product crtProduct: products) {
        if (crtProduct.getColor().equals(color))
            selectedElements.add(crtProduct);
    }
    return selectedElements;
}
```

NO Copy&Paste... and slightly change!!!

```java
public Collection<Product> selectByProducer(String producer) {
    Collection<Product> selectedElements = new ArrayList<Product>();

    for(Product crtProduct: products) {
        if (crtProduct.getProducer().equals(producer))
            selectedElements.add(crtProduct);
    }
    return selectedElements;
}
```

… a better solution
Inside the Warehouse class!!!

```
public Collection<Product> selectByCondition(Condition condition) {
    Collection<Product> selectedElements = new ArrayList<Product>();

    for(Product crtProduct: products) {
        if (condition.test(crtProduct))
            selectedElements.add(crtProduct);
    }
    return selectedElements;
}
```


ColorCondition
color String
test(Product) boolean
Condition
test(Product) boolean
DescriptionCondition
description String
test(Product) boolean
AndCondition
condition1 Condition
condition2 Condition
test(Product) boolean
ProducerCondition
producer String
test(Product) boolean

```
Condition c1 = new ColorCondition("red");
Condition c2 = new ProducerCondition("EBS");
Condition c3 = new DescriptionCondition("paper");

Condition cc = new AndCondition(c1, c2);
Condition ccc = new AndCondition(c3, cc);

System.out.println(myWarehouse.selectByCondition(ccc));
```

... more classes like OrCondition, NotCondition will be added!!!

... an anonymous class!!!

```
System.out.println(myWarehouse.selectByCondition(new Condition () {
    public boolean test(Product product) {
        return product.getColor().equals("red");
    }}
));
```

=> increased amount of code / verbose

=> reading is hampered

... use lambda expressions!!!

```
System.out.println(myWarehouse.selectByCondition(
        (Product product) ->  product.getColor().equals("red")));
```

=> pass behaviour/code as short as possible, without many boilerplate code

=> anonymous functions passed as arguments to methods

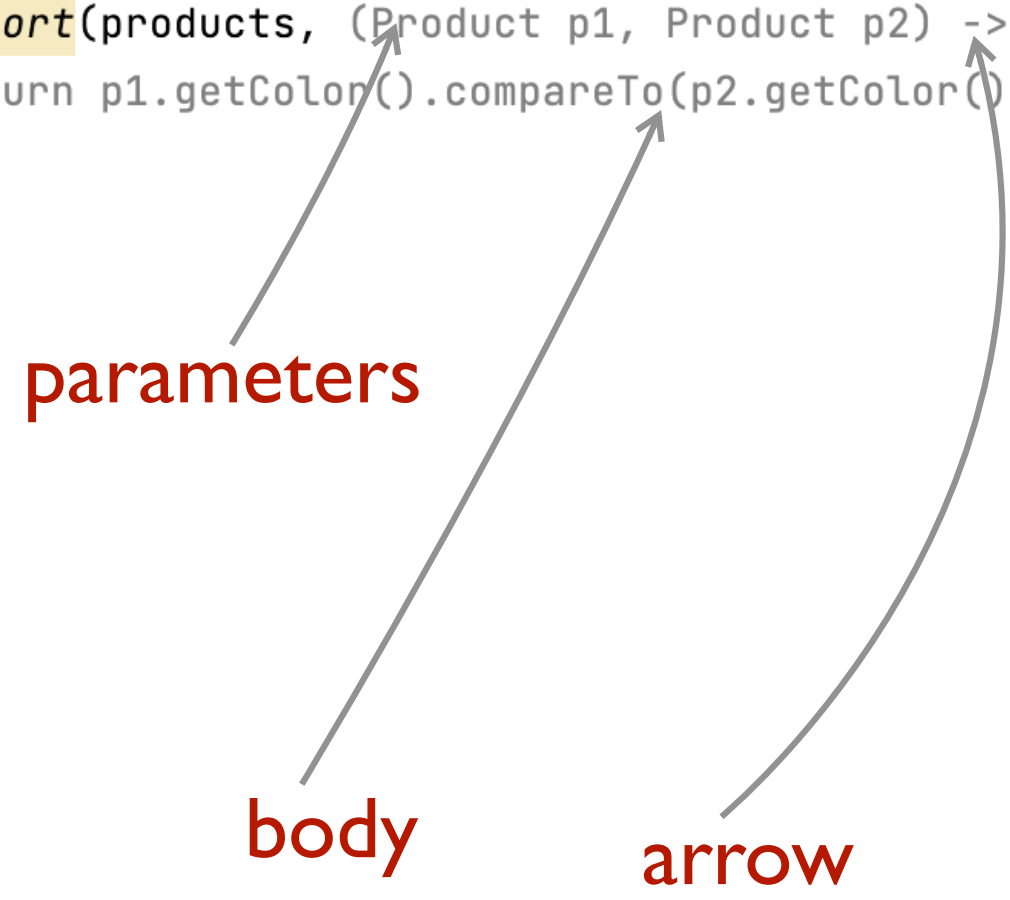
=> behaviour is sent as a parameter to a function!!!

Sorting arrays

```
List<Product> products = new ArrayList<>();


Collections.sort(products, new Comparator<Product>() {
    public int compare(Product p1, Product p2) {
        return p1.getColor().compareTo(p2.getColor());
    }}
);


Collections.sort(products, (Product p1, Product p2) ->
        { return p1.getColor().compareTo(p2.getColor()); });

Collections.sort(products, (Product p1, Product p2) ->
        p1.getColor().compareTo(p2.getColor()));

Collections.sort(products, (p1, p2) ->
        p1.getColor().compareTo(p2.getColor()));
```

Lambda expressions

```
Collections.sort(products, (Product p1, Product p2) ->
        { return p1.getColor().compareTo(p2.getColor()); });
```



=> (parameters) -> expression; expression is considered the return value of Lambda

=> (parameters) -> { statements; }

Where can Lambda expressions be used?

java.util

Interface Comparator<T>

https://docs.oracle.com/javase/8/docs/api/java/util/Comparator.html

Type Parameters:

`T - the type of objects that may be compared by this comparator`

All Known Implementing Classes:

`Collator, RuleBasedCollator`

Functional Interface:

`This is a functional interface and can therefore be used as the a`

All Methods	Static Methods	Instance Methods	Abstract Methods	Default Methods

Modifier and Type	Method and Description
int	`compare(T o1, T o2)` Compares its two arguments for order.

A functional interface is an interface that contains exactly one abstract method and a Lambda is associated with a functional interface.

```
Comparator<Product> myC;
myC = (p1, p2) -> p1.getColor().compareTo(p2.getColor());
```

```
public interface Condition {
    public boolean test(Product product);
}
```

```
Condition c1;
c1 = (Product product) ->  product.getColor().equals("red");

Condition c2;
c2 = (Product product) ->  {
    return product.getColor().equals("red");
};
```

java.util.function

Interface Predicate<T>

https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html

Type Parameters:

`T - the type of the input to the predicate`

Functional Interface:

```
This is a functional interface and can therefore be used as
          the assignment target for a lambda expression or method reference.
```

`boolean`	**`test`**`(T t)` Evaluates this predicate on the given argument.

```
Predicate<Product> p1;
p1 = (Product product) ->  product.getColor().equals("red");
```

Functional interfaces - examples

```java
interface Calculator {
    int calculate(int a, int b);
}
```

```java
interface ScientificCalculator extends Calculator {
    double calculate(double a, double b);
}
```

Functional interfaces - examples

```java
interface Printer {
    void print(String message);
}
```

```java
interface FormattablePrinter extends Printer {
    String format(String message);
}
```

Functional interfaces - examples

```java
interface Printer {
    void print(String message);
}
```

```java
interface FormattablePrinter extends Printer {
    default String format(String message) {
        return "XXXX";
    }
}
```

Lambda expressions let you provide the implementation of the abstract method of a functional interface directly inline and *treat the whole expression as an instance of a functional interface*.

Functional interfaces are useful because the signature of the abstract method can describe the signature of a lambda expression. The signature of the abstract method of a functional interface is called a function descriptor.

Lambda expressions - examples

```
interface Calculator {
    int calculate(int a, int b);
}
```

```
Calculator c1 = (int a, int b) -> { return a + b; };

Calculator c2= (int a, int b) -> a + b;

Calculator c3= (a, b) -> a + b;
```

```
Condition cond;
cond = (Product product) ->  product.getColor().equals("red");
```

```
Predicate<Product> p1;
p1 = (Product product) ->  product.getColor().equals("red");
```

Warehouse extended

```
public Collection<Product> selectByCondition(Condition condition) {
    Collection<Product> selectedElements = new ArrayList<Product>();

    for(Product crtProduct: products) {
        if (condition.test(crtProduct))
            selectedElements.add(crtProduct);
    }
    return selectedElements;
}

public Collection<Product> selectByCondition(Predicate<Product> condition) {
    Collection<Product> selectedElements = new ArrayList<Product>();

    for(Product crtProduct: products) {
        if (condition.test(crtProduct))
            selectedElements.add(crtProduct);
    }
    return selectedElements;
}
```

```
System.out.println(myWarehouse.selectByCondition(
        (Product product) ->  product.getColor().equals("red")));
```

Compilation ERROR

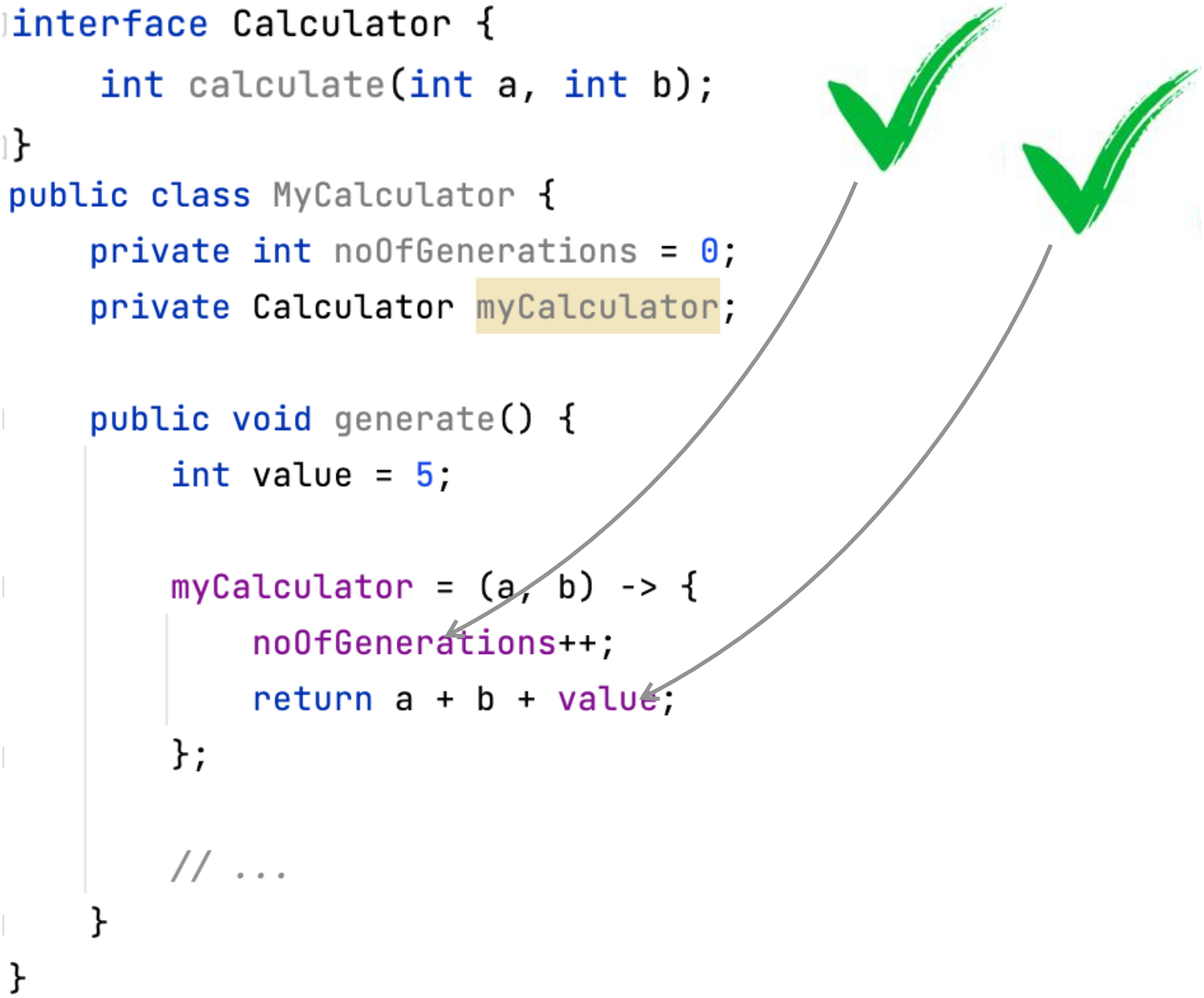
java: reference to selectByCondition is ambiguous
both method selectByCondition(Condition) in Warehouse
and method
selectByCondition(java.util.function.Predicate<Product>) in
Warehouse match

```
System.out.println(myWarehouse.selectByCondition(
        (Predicate<Product>)(Product product) ->
                product.getColor().equals("red")
        ));

System.out.println(myWarehouse.selectByCondition(
        (Condition)(Product product) ->
                product.getColor().equals("red")
));
```

Local variables inside lambdas

Lambdas
=> can freely use variables instance and static variables without restrictions

=> local variables have to be explicitly declared final or be effectively final


interface Calculator {
int calculate(int a, int b);
}
public class MyCalculator {
private int noOfGenerations = 0;
private Calculator myCalculator;
public void generate() {
int value = 5;
myCalculator = (a, b) -> {
noOfGenerations++;
return a + b + value;
};
// ...
}
}

Compilation ERROR

java: local variables referenced from a lambda expression must be final or effectively final

```
public class MyCalculator {
    private int noOfGenerations = 0;
    private Calculator myCalculator;

    public void generate() {
        int value = 5;

        myCalculator = (a, b) -> {
            noOfGenerations++;
            return a + b + value++;
        };

        // ...
    }
}
```

Method references

=> let you create a lambda expression from an existing method implementation

Lambda	Method reference (similar)
(String s) -> Integer.parseInt(s)	Static method reference Integer::parseInt
(String s) -> s.length()	Instance Method Reference of a parameter String::length
() -> product.getColor()	Instance Method Reference of an existing object product::getColor

Streams

=> let you manipulate collections of data in a declarative way

=> an enhancement of the Java API (Java 8)

Example

```
List<Product> products = new ArrayList<>();
```

Requirement: get the names of the distinct producers sorted descending

```
//get the distinct producers
Collection<String> producers = new HashSet<>();
for (Product p: products) {
    producers.add(p.getProducer());
}


//sort descending
List<String> sortedProducers = new ArrayList(producers);
Collections.sort(sortedProducers, new Comparator<String>() {
    public int compare(String name1, String name2) {
        return name1.compareTo(name2) * (-1);
    }
});
```

The provided code is correct in terms of functionality. It creates a copy of the **`producers`** list using the **`ArrayList`** constructor and then sorts the **`sortedProducers`** list in reverse order using a custom **`Comparator`** implementation.

However, there is a more concise way to achieve the same result using lambda expressions.

```
producers = products.stream().
                map((p) -> p.getProducer()).
                distinct().
                sorted((name1, name2) -> name1.compareTo(name2) * (-1)).
                collect(Collectors.toList());
```

```
producers = products.stream().
                map(Product::getProducer).
                distinct().
                sorted((name1, name2) -> name1.compareTo(name2) * (-1)).
                collect(Collectors.toList());
```

map -> it's used to transform each element in a stream into another element according to a given function. The result is a new stream containing the transformed elements.

Requirement: get the products manufactured by EBS

```
List<Product> selectedProducts;
selectedProducts = products.stream().
        filter((p) -> p.getProducer().equals("EBS")).
        collect(Collectors.toList());
```

filter -> it's used to select elements from a stream based on a specified condition. The result is a new stream containing only the elements that satisfy the condition.

Requirement: find if the warehouse contains products manufactured by XXX

```
boolean contains = products.stream().
        anyMatch((p) -> p.getProducer().equals("XXX"));
```

anyMatch -> it's used to check whether any element in a stream satisfies a given condition (predicate). It is a terminal operation and returns a boolean value indicating whether at least one element in the stream matches the specified condition.

Requirement: get the product having the highest price

```
Product highestPriced = products.stream().
        reduce((p1, p2) ->
            p1.getPrice() > p2.getPrice() ? p1 : p2).
        orElse(null);
```

reduce -> it's used to combine the elements of a stream into a single value using a specified operation.

Requirement: get the numbers of characters read from a file

CR read the content of a file and place it inside a collection of String

To read the content of a file and store it in a collection of strings, you can use the **`BufferedReader`** class along with a **`List`** implementation such as **`ArrayList`**. Here's an example:

```java
//BEGIN OF GENERATED CODE with ChatGPT May 12 Version
public class ReadFileIntoCollection {
    public static void main(String[] args) {
        String filePath = "/Users/Cristina/Desktop/T2.txt"; // Replace with the actual file path

        List<String> lines = new ArrayList<>();

        try (BufferedReader reader = new BufferedReader(new FileReader(filePath))) {
            String line;

            while ((line = reader.readLine()) != null) {
                lines.add(line);
            }
            System.out.println("File content has been read into the collection.");
        } catch (IOException e) {
            System.out.println("An error occurred while reading the file: " + e.getMessage());
        }

        // Print the lines from the collection
        for (String line : lines) {
            System.out.println(line);
        }
    }
    //END OF GENERATED CODE
```

```java
// Print the lines from the collection
for (String line : lines) {
    System.out.println(line);
}

//END OF GENERATED CODE

//NON GENERATED CODE

lines.stream().forEach(line -> System.out.println(line));

lines.stream().forEach(System.out::println);



lines.forEach(line -> System.out.println(line));

lines.forEach(System.out::println);
```

Requirement: get the numbers of characters read from a file

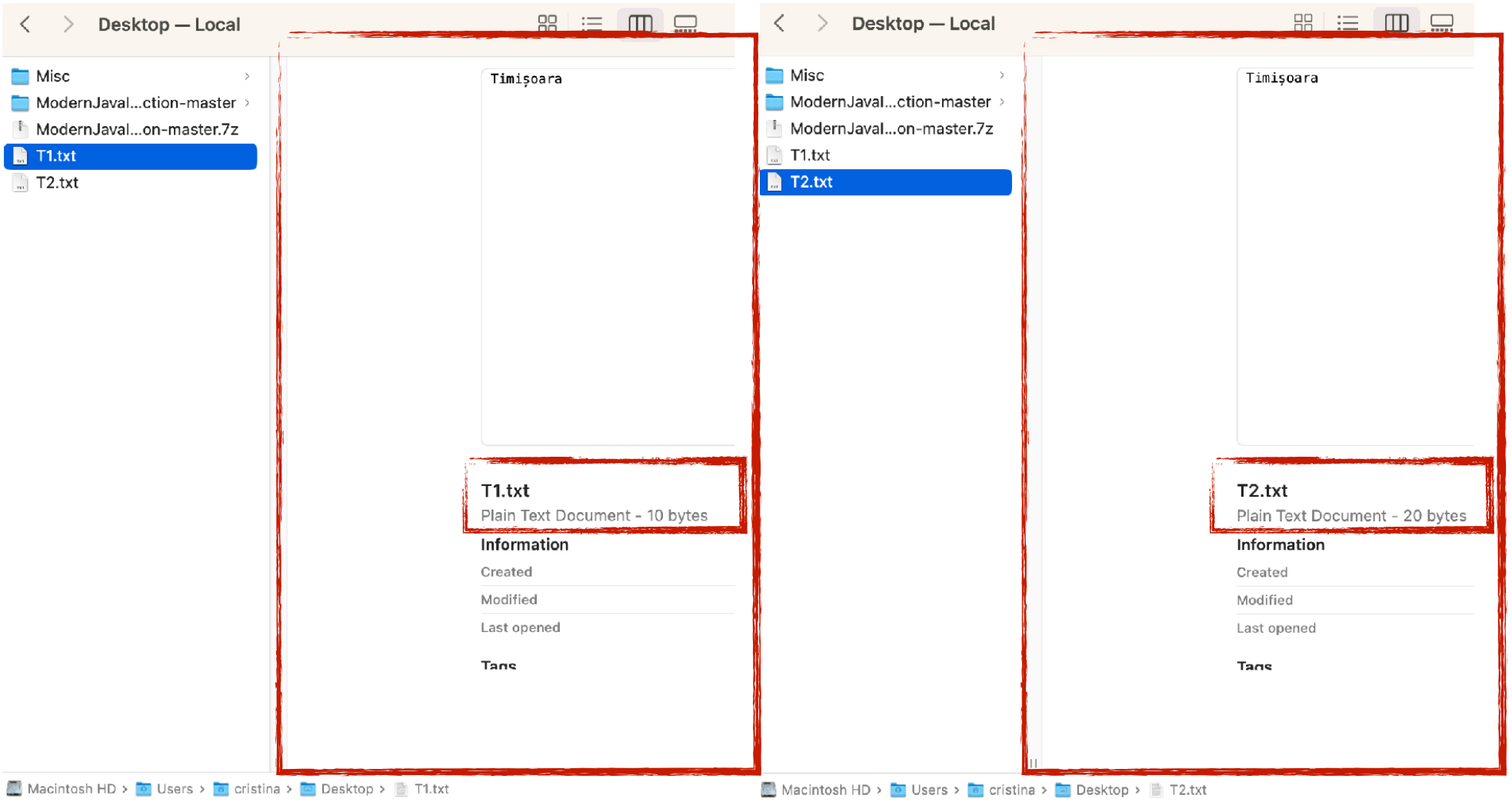
```
List<String> lines = new ArrayList<>();


int numberOfReadCharacters = lines.stream().
        map(line -> line.length()).
        reduce(0, (l1, l2) -> l1 + l2);
```

Requirement: get the numbers of characters read from a file

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


int numberOfReadCharacters = lines.stream().
        map(String::length).
        reduce(0, Integer::sum);
```

Results: same content…but different output!!!


```
        File content has been read into the collection.
T1.txt  Timișoara
        9

        File content has been read into the collection.
T2.txt  ␀␀T␀i␀m␀i␀␙␂o␀a␀r␀a␀
        20
```



Unicode, UTF-8, and ASCII

https://medium.com/@apiltamang/unicode-utf-8-and-ascii-encodings-made-easy-5bfbe3a1c45a

Unicode is a character encoding standard that aims to provide a unique numerical value (code point) for every character used in writing across the world's different writing systems, languages, and scripts. It enables computers to represent and manipulate text from various languages and symbols consistently and uniformly, regardless of the platform, software, or language being used.

Unicode addresses the limitations of older character encoding schemes like ASCII, which only supported a limited set of characters primarily used in the English language. As global communication and software development became more diverse, it became evident that a more comprehensive and flexible character encoding system was needed.

Unicode assigns a unique numeric value to each character, including letters, digits, punctuation marks, symbols, and even non-visible control characters. These numeric values are called "code points." Unicode defines a wide range of code points that accommodate characters from virtually all languages and scripts, as well as special symbols and formatting characters.

Unicode is designed to be extensible, allowing new characters to be added as needed. The most commonly used character encoding forms of Unicode are UTF-8, UTF-16, and UTF-32, which determine how the code points are represented using bytes.

```java
List<String> lines = new ArrayList<>();

try (BufferedReader reader = new BufferedReader(new FileReader(filePath,
        Charset.forName("UTF-16")))) {
    String line;

    while ((line = reader.readLine()) != null) {
        lines.add(line);
    }
    System.out.println("File content has been read into the collection.");
} catch (IOException e) {
    System.out.println("An error occurred while reading the file: " + e.getMessage());
}
```

T1.txt

```
File content has been read into the collection.
咻浩鄨激牡
5
```

T2.txt

```
File content has been read into the collection.
Timișoara
9
```

Requirement: get the products sold by each existing producer in the warehouse!

=> which are the producers?
=> which is the suitable data structure for keeping the result?
=> populate the data structure?

WHAT versus HOW

Version I

```
List<Product> products = new ArrayList<>();
// . . .

Map<String, List<Product>> productsByProducers;
productsByProducers = new HashMap<>();

for (Product p : products) {
    String producer = p.getProducer();

    List<Product> productsOfProducer;
    productsOfProducer = productsByProducers.get(producer);

    if (productsOfProducer == null) {
        productsOfProducer = new ArrayList<>();
        productsByProducers.put(producer, productsOfProducer);
    }

    productsOfProducer.add(p);
}
```

Version 2...

```
productsByProducers = products.stream().
        collect(Collectors.groupingBy(Product::getProducer));
```

```
public final class Collectors
extends Object
```

Implementations of `Collector` that implement various useful reduction operations, such as accumulating elements into collecti

The following are examples of using the predefined collectors to perform common mutable reduction tasks:

```
// Accumulate names into a List
List<String> list = people.stream().map(Person::getName).collect(Collectors.toList());

// Accumulate names into a TreeSet
Set<String> set = people.stream().map(Person::getName).collect(Collectors.toCollection(TreeSet::new));

// Convert elements to strings and concatenate them, separated by commas
String joined = things.stream()
                      .map(Object::toString)
                      .collect(Collectors.joining(", "));

// Compute sum of salaries of employee
int total = employees.stream()
                     .collect(Collectors.summingInt(Employee::getSalary)));

// Group employees by department
Map<Department, List<Employee>> byDept
    = employees.stream()
               .collect(Collectors.groupingBy(Employee::getDepartment));

// Compute sum of salaries by department
Map<Department, Integer> totalByDept
    = employees.stream()
               .collect(Collectors.groupingBy(Employee::getDepartment,
                                              Collectors.summingInt(Employee::getSalary)));

// Partition students into passing and failing
Map<Boolean, List<Student>> passingFailing =
    students.stream()
            .collect(Collectors.partitioningBy(s -> s.getGrade() >= PASS_THRESHOLD));
```

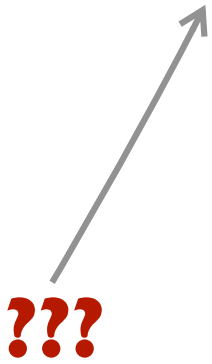
Requirement: get the sum of all prices!

```
double sum = products.stream().
        map(Product::getPrice).
        reduce((float)0.0, (p1, p2) -> p1 + p2);

sum = products.stream().
        map(Product::getPrice).
        reduce((float)0.0, Float::sum);

sum = products.stream().
        collect(Collectors.summingDouble(Product::getPrice));

sum = products.stream().mapToDouble(Product::getPrice).sum();
```

REQ Get a duplicate of the warehouse.

=> means implementing cloning that can be shallow or deep

```
// create a copy of the warehouse
Warehouse clonedWarehouse = myWarehouse.clone();
System.out.println(myWarehouse == clonedWarehouse);
System.out.println(myWarehouse.getProducts()
        == clonedWarehouse.getProducts());
```

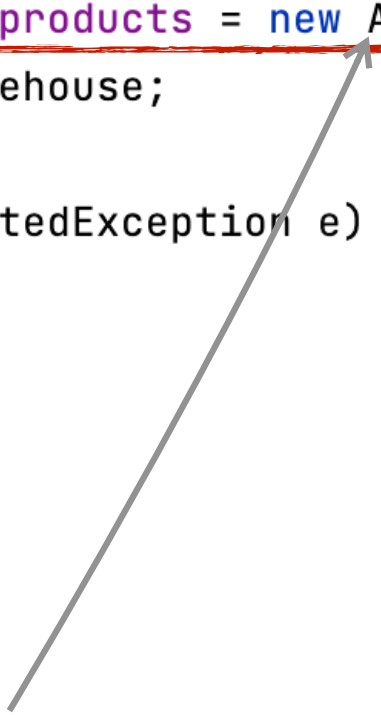


???

```
@Override
public Warehouse clone() {
    return this.shallowClone();
    //return this.deepClone();
}
```

```
public Warehouse shallowClone() {
    try {
        return (Warehouse)super.clone();
    }
    catch(CloneNotSupportedException e) {
        return null;
    }
}
```

```java
public Warehouse deepClone() {
    try {
        Warehouse clonedWarehouse = (Warehouse)super.clone();
        clonedWarehouse.products = new ArrayList<>(products);
        return clonedWarehouse;
    }
    catch(CloneNotSupportedException e) {
        return null;
    }
}
```



products are the same

```java
public Warehouse deepClone() {
    try {
        Warehouse clonedWarehouse = (Warehouse)super.clone();

        // Deep clone the products list
        clonedWarehouse.products = new ArrayList<>(products.size());
        for (Product product : products) {
            clonedWarehouse.products.add((Product)product.clone());
        }
        return clonedWarehouse;
    }
    catch(CloneNotSupportedException e) {
        return null;
    }
}
```

add clone() inside Product

REQ Get a duplicate of the warehouse without using clone().

Hint Use a copy constructor Warehouse(Warehouse warehouse).

Do you think the copy constructor is better than clone?

REQ We have inside a method a reference to an object whose type is String, as it follows:

String s = “MyString";

Present the changes we have to perform in order to modify the content of the existing String, from "MyString" to "MyNEWString".

REQ Remove the code duplication between the next two derived classes.

```
abstract class CompoundExpression extends Expression {
        //references'values are set in the constructor
        private Expression e1, e2;

        public Expression getLExpression() {
                return e1;
        }

        public Expression getRExpression() {
                return e2;
        }
}

class AddExpression extends CompoundExpression {
        . . .
        public String toString() {
                return "(" + this.getLExpression() + "+" + this.getRExpression() + ")";
        }
}

class MultiplyExpression extends CompoundExpression {
        . . .
        public String toString() {
                return "(" + this.getLExpression() + "*" + this.getRExpression() + ")";
        }
}
```