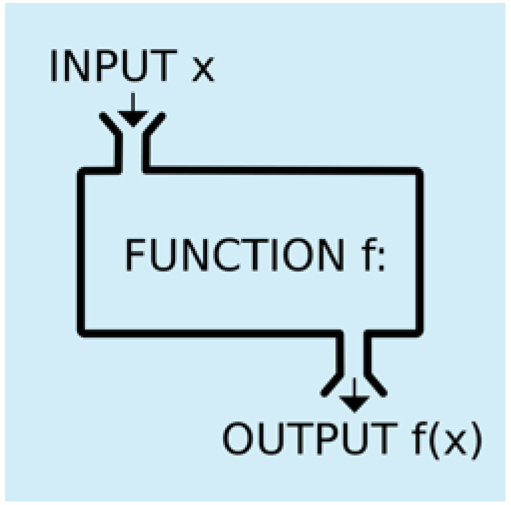
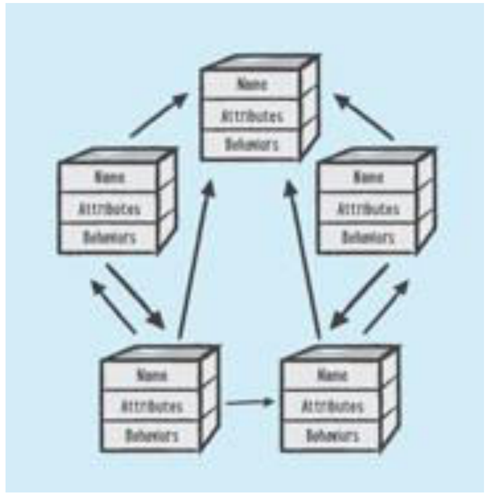
Lambda expressions & streams

What are Lambda expressions?



Functional ⇔ Object Oriented



Lambda expressions are functions that can be passed around **as arguments**.

Using lambdas, you can create "anonymous methods" that **implement functional interfaces** with a more concise syntax.

Lambdas have become increasingly popular with the movement towards more **functional programming** (<-> Object Oriented): A Lambda is more like a function than a method. This is because a method belongs to a class whereas a Lambda doesn't. But just like a method, a Lambda accepts a list of parameters, has a body and can return a value.

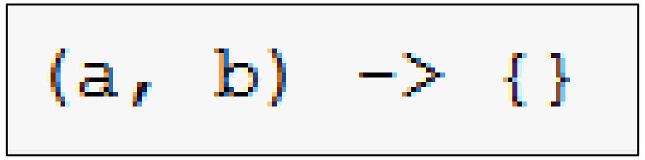
Why use Lambda expressions?

- A more **concise** syntax: lambda expressions allow to implement a functional interface with fewer lines of code
- Lambdas make it easier to work with collections and **streams**

Lambda functions

- **Video**

Lambda Syntax



```
(a, b) -> {}
```

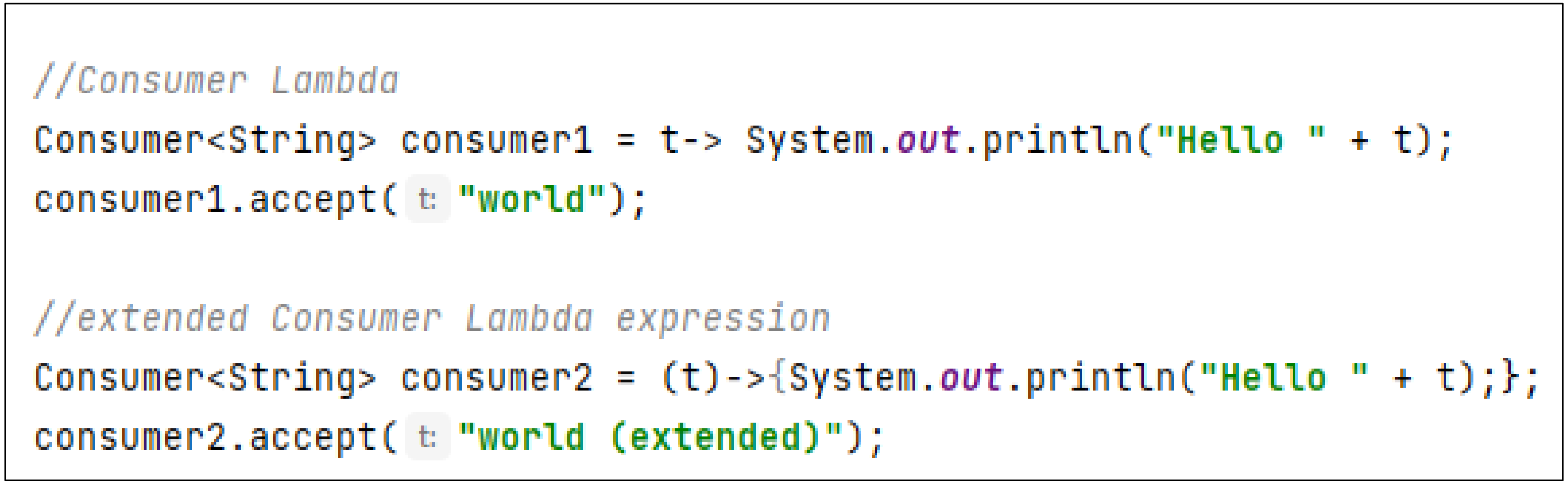
- **Argument List (a,b)**:
  The argument list is a comma-separated list of parameters enclosed in parentheses. These arguments usually don't require type declarations as their types can be inferred by the compiler. The parentheses () are optional if you only have a single argument.
- **2) Arrow token ->**
- This is the syntax for lambda expressions and signifies the passing of the parameters to the body.
- **3) Body {}**
- The body is a code block that represents the implementation of a functional interface. If the body is a single statement, the expression will evaluate and return the result. In this case, the brackets {} are optional. If the body is more than a single statement, you must use brackets and return the result.
- **Expressions are limited**. They have to immediately return a value, and they cannot contain variables, assignments or statements such as `if` or `for`.

The java.util.function package

- Java provides a set of reusable, generic, functional interfaces that are *ready for use out of the box* in the package java.util.function:

Name	Input	Output	Example
Consumer	Object	void	i -> System.out.println(i);
Supplier	void	Object	() -> new Integer(1);
Function	Object	Object	i -> i.toString();
Predicate	Object	Boolean	i -> i.equals('a');
Comparator	Object Object	Int	(i1, i2) -> i1.compareTo(i2);
UnaryOperator	Object	Object	(String s) -> s + '!';
BiPredicate	Object, Object	Boolean	(i1,i2) -> i1.equals(i2);
BiFunction	Object, Object	Object	(i1,i2) -> i1 + i2;

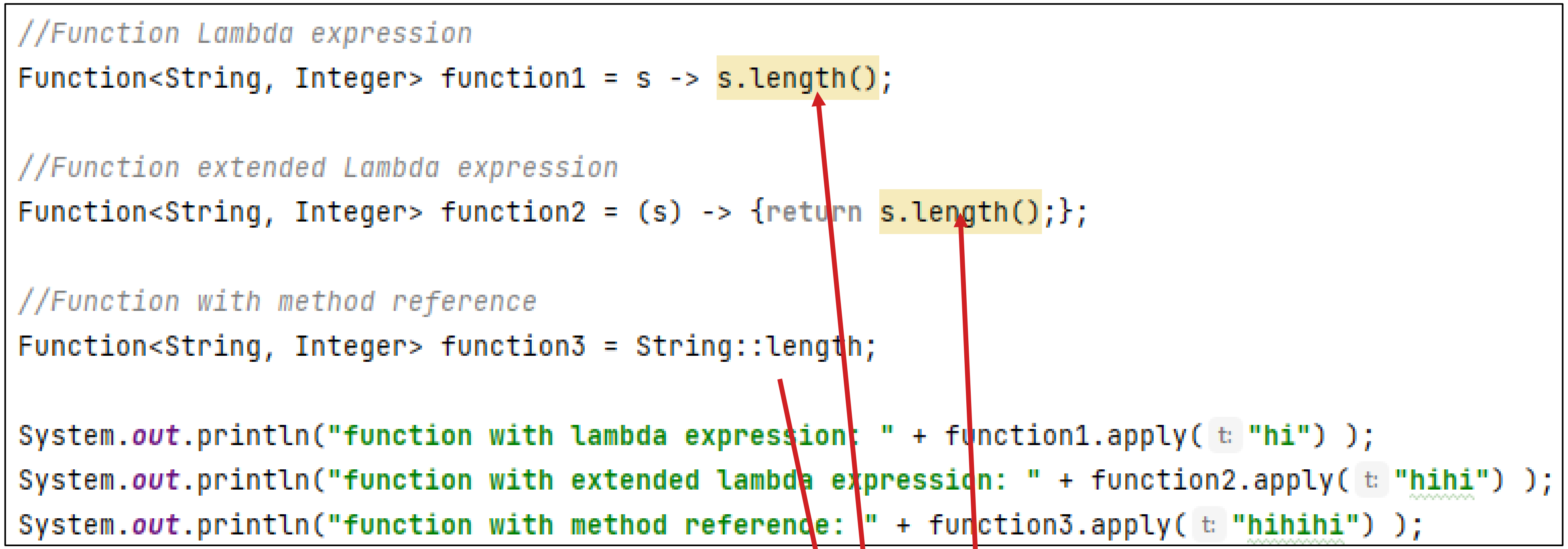
Consumer Example



```java
//Consumer Lambda
Consumer<String> consumer1 = t-> System.out.println("Hello " + t);
consumer1.accept( t: "world");

//extended Consumer Lambda expression
Consumer<String> consumer2 = (t)->{System.out.println("Hello " + t);};
consumer2.accept( t: "world (extended)");
```

- parentheses () and brackets {} are not mandatory if you have only one argument / statement

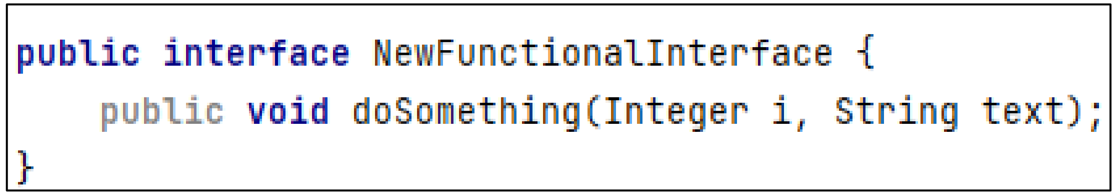
Function Example



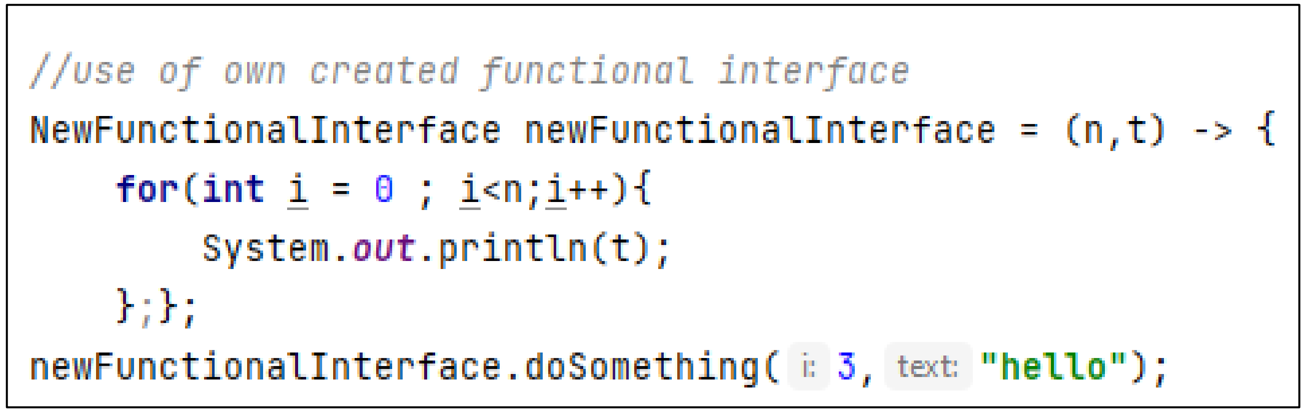
```
//Function Lambda expression
Function<String, Integer> function1 = s -> s.length();

//Function extended Lambda expression
Function<String, Integer> function2 = (s) -> {return s.length();};

//Function with method reference
Function<String, Integer> function3 = String::length;

System.out.println("function with lambda expression: " + function1.apply( t: "hi") );
System.out.println("function with extended lambda expression: " + function2.apply( t: "hihi") );
System.out.println("function with method reference: " + function3.apply( t: "hihihi") );
```

- The general syntax for a method reference is: `Object :: method`
- With a method reference, you don't need to worry about passing arguments. In the above example, the compiler can infer the argument based on the method definition of *length()*.
- IntelliJ suggests to use a method reference wherever this is possible. That is why the code is yellow…

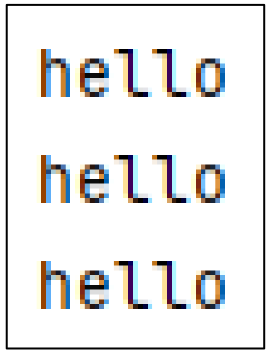
Example of own created functional interface



```java
public interface NewFunctionalInterface {
    public void doSomething(Integer i, String text);
}
```



```java
//use of own created functional interface
NewFunctionalInterface newFunctionalInterface = (n,t) -> {
    for(int i = 0 ; i<n;i++){
        System.out.println(t);
    };};
newFunctionalInterface.doSomething( i: 3, text: "hello");
```



```
hello
hello
hello
```

A functional interface in Java is **an interface that contains only one single abstract (unimplemented) method**

Lambdas in collection-methods - Examples

```
ArrayList<String> list = new ArrayList<>();
list.add("a");
list.add("b");
list.add("c");
System.out.println("first iteration");
list.forEach(x -> System.out.println(x));
list.removeIf(n -> n.equals("b"));
System.out.println("second iteration");
list.forEach(x -> System.out.println(x));
list.replaceAll(n->n+"!");
System.out.println("third iteration");
list.forEach(System.out::println);
```

```
first iteration
a
b
c
second iteration
a
c
third iteration
a!
c!
```

- In java api doc 17:

void	forEach(Consumer<? super E> action)
void	replaceAll(UnaryOperator<E> operator)
boolean	removeIf(Predicate<? super E> filter)

  - *Consumer, UnaryOperator* and *Predicate* = functional interfaces (in the java.util.function-package) that can be implemented using a Lambda expression
  - *super E* = ArrayList implementation. In this case “String”
- Also System.out.println(x) can be replaced bij a method reference.

Lambdas in collection-methods – Examples2

```
ArrayList<Integer> numbers= new ArrayList<>();
numbers.add(6);
numbers.add(5);
numbers.add(1);
numbers.add(9);
//if j-i = output, the list will be sorted in descending order
numbers.sort((i, j)->j-i);
```

```
9
6
5
1
```

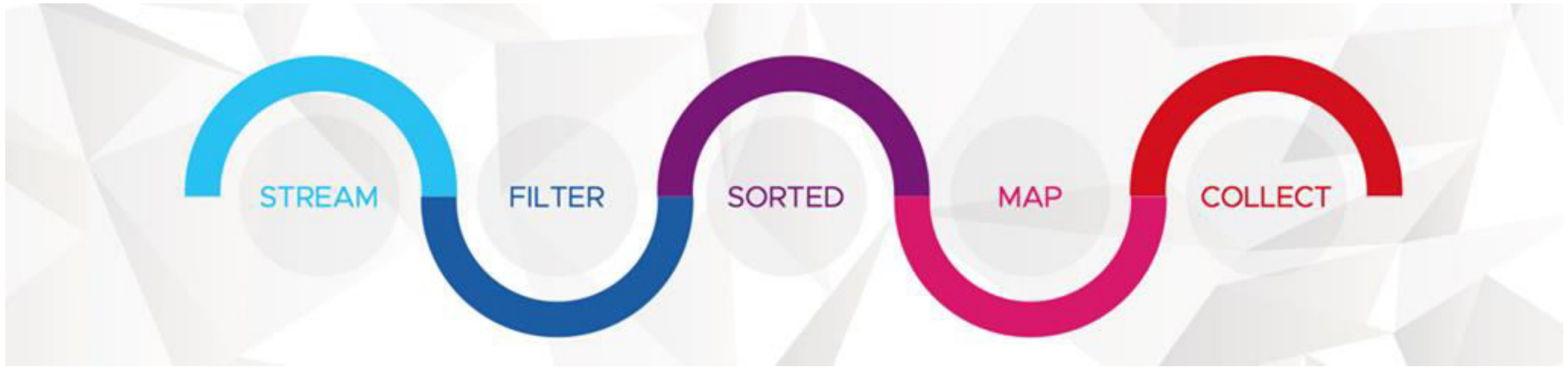
- In java api doc 17:

```
void sort(Comparator<? super E> c)
```

  - a Comparator-implementation returns an Integer (j-i) which is used in the sort-method to define the sequence.
  - Alternative:

```
numbers.sort((i, j)->j.compareTo(i)) ;
```

Java Streams



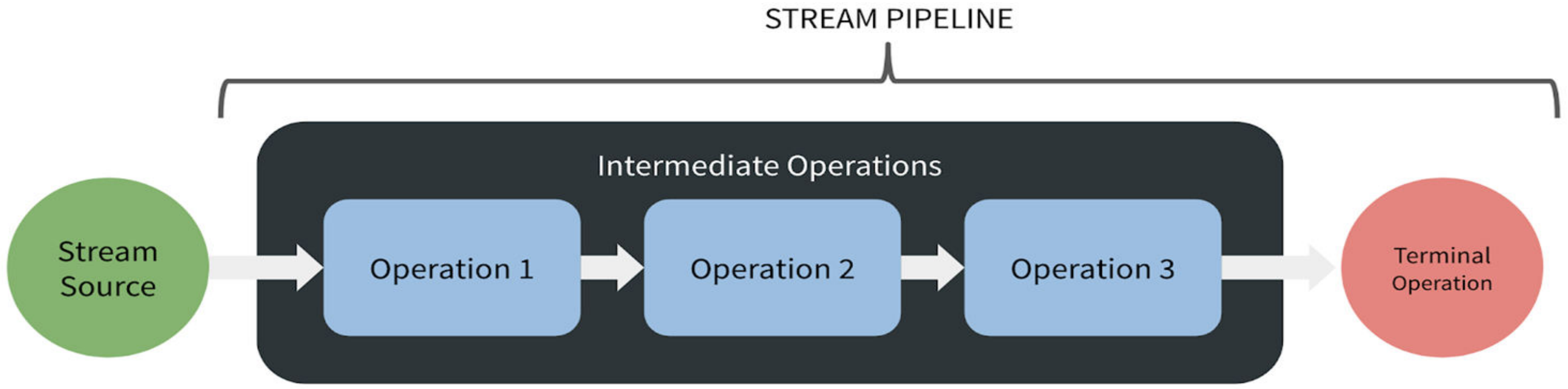
We can see Stream as a “data flow” abstraction which allows us to transform or manipulate the data it is containing.

Unlike the other collections, a Stream doesn’t allow us a direct access to the element it contains. Although if you want to access the elements, we can always transform the stream into one of the collections in Java and fulfill our purpose.

List vs Stream

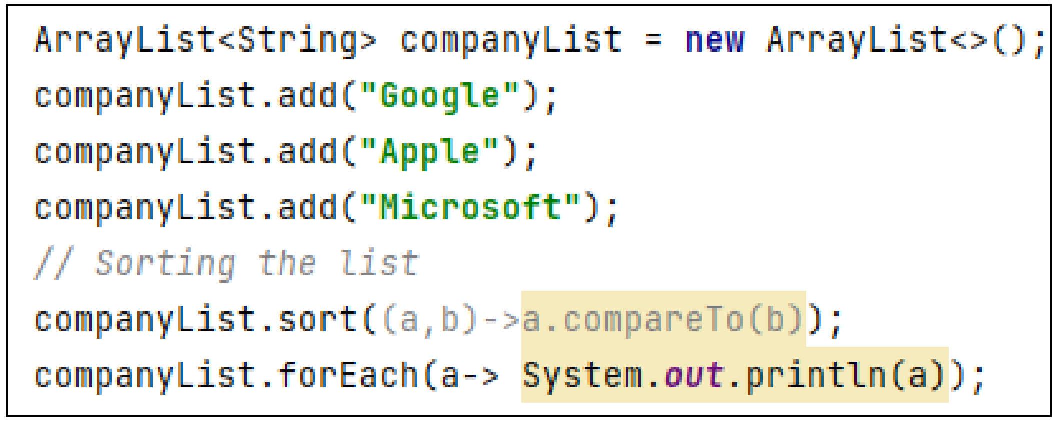
List	Stream
Collection	<u>NO</u> Collection
Stores/holds data	Does not store data, it operates on the collection
Several iterations	1x iteration
Lists are modifiable i.e one can easily add to or remove elements from collections.	Streams are not modifiable i.e one can't add or remove elements from streams
Lists are iterated externally using loops	Streams are iterated internally by just mentioning the operations.
Set, List, Map, ...	filter, map, collect, ...

Streams

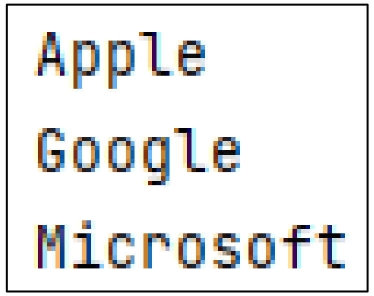


- Streams don’t change the original data structure, they only provide the result as per the **pipelined methods** and don’t change the original data structure.
- Each intermediate operation is lazily executed and returns a stream as a result, hence various intermediate operations can be pipelined.
- **Terminal operations** mark the end of the stream and return the result.

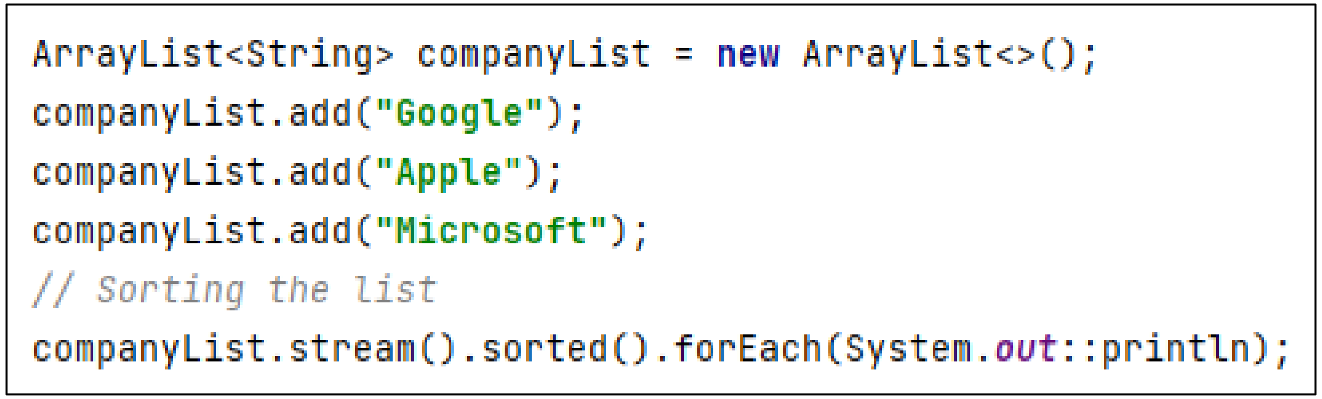
Collection-methods vs Stream– Examples



```
ArrayList<String> companyList = new ArrayList<>();
companyList.add("Google");
companyList.add("Apple");
companyList.add("Microsoft");
// Sorting the list
companyList.sort((a,b)->a.compareTo(b));
companyList.forEach(a-> System.out.println(a));
```

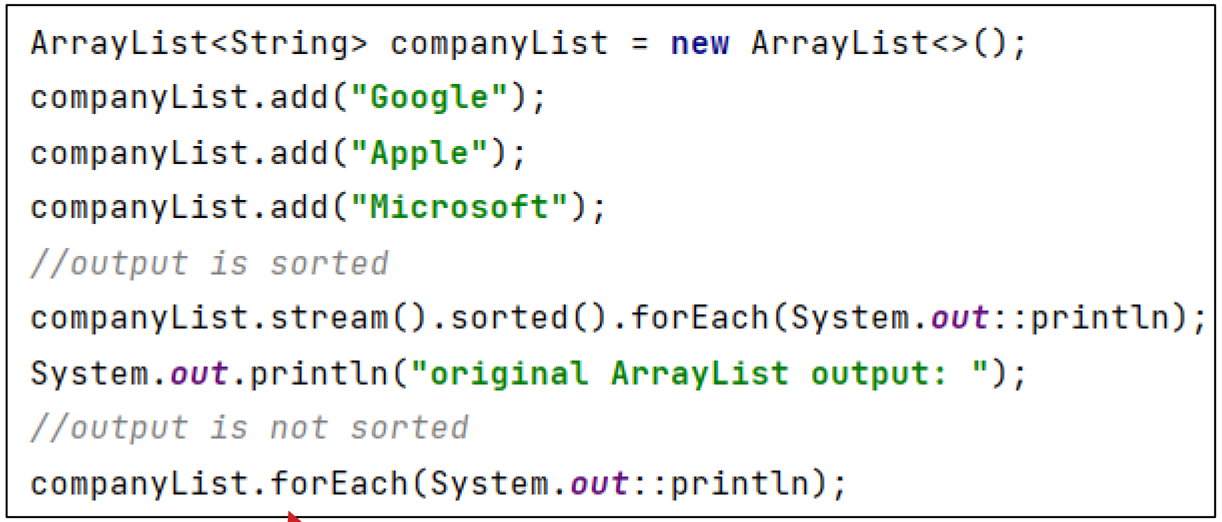


```
Apple
Google
Microsoft
```

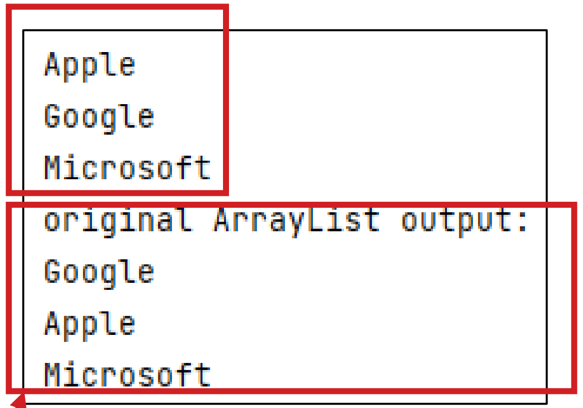


```
ArrayList<String> companyList = new ArrayList<>();
companyList.add("Google");
companyList.add("Apple");
companyList.add("Microsoft");
// Sorting the list
companyList.stream().sorted().forEach(System.out::println);
```

Stream – Example1



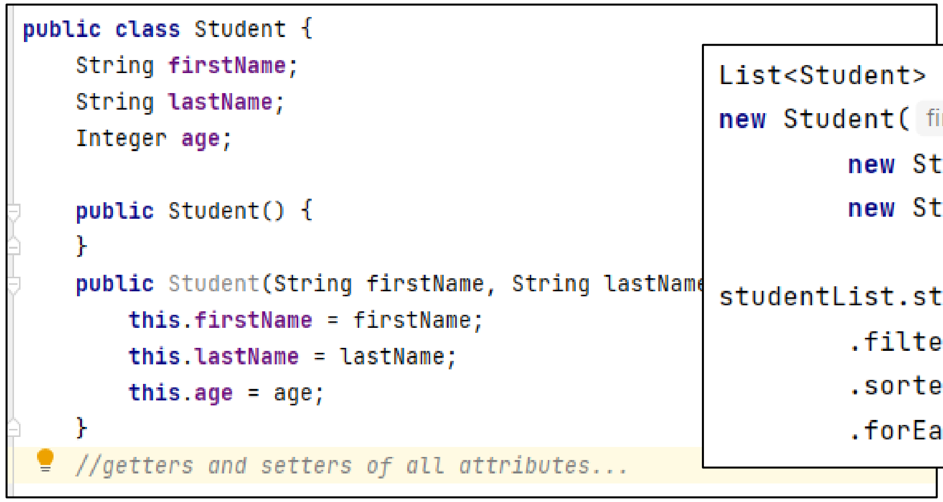
```java
ArrayList<String> companyList = new ArrayList<>();
companyList.add("Google");
companyList.add("Apple");
companyList.add("Microsoft");
//output is sorted
companyList.stream().sorted().forEach(System.out::println);
System.out.println("original ArrayList output: ");
//output is not sorted
companyList.forEach(System.out::println);
```



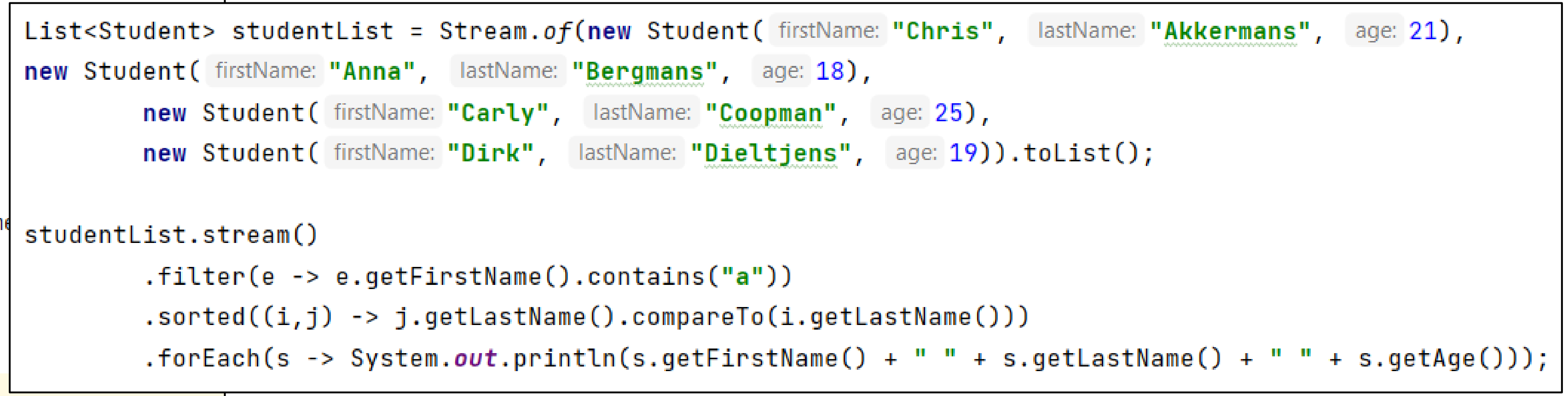
```
Apple
Google
Microsoft
original ArrayList output:
Google
Apple
Microsoft
```

- *stream()* and *sorted()* are **intermediate pipeline operations** and return a stream
- forEach is a **terminal operation**
- content of companyList has NOT changed

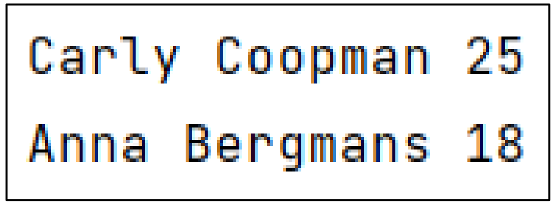
Stream – Example2



```
public class Student {
    String firstName;
    String lastName;
    Integer age;

    public Student() {
    }
    public Student(String firstName, String lastName
        this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;
    }
    //getters and setters of all attributes...
```



```
List<Student> studentList = Stream.of(new Student( firstName: "Chris",  lastName: "Akkermans",  age: 21),
new Student( firstName: "Anna",  lastName: "Bergmans",  age: 18),
        new Student( firstName: "Carly",  lastName: "Coopman",  age: 25),
        new Student( firstName: "Dirk",  lastName: "Dieltjens",  age: 19)).toList();

studentList.stream()
        .filter(e -> e.getFirstName().contains("a"))
        .sorted((i,j) -> j.getLastName().compareTo(i.getLastName()))
        .forEach(s -> System.out.println(s.getFirstName() + " " + s.getLastName() + " " + s.getAge()));
```



```
Carly Coopman 25
Anna Bergmans 18
```

- *Stream.of()* : to create a stream
- *toList():* to create a **new** List
- *sorted():* to sort the stream according to the natural order.
- *sorted(Comparator<? super T> comparator):* to use a different order and/or define a different element to sort on

Other interesting stream-methods

- Intermediate operations (https://www.javacodegeeks.com/2020/04/java-8-stream-intermediate-operations-methods-examples.html)

```
 .filter(Predicate)
 .map(Function)
 .peek(Consumer)
 .skip(Long)
 .sorted(Comparator)
```

- Terminal operations:

```
 .collect(Collectors.toList());
 .findFirst();
 .anyMatch(Predicate);
 .noneMatch(Predicate);
 .count();
```

Conclusion

- Lambda expressions are very interesting to use because:
  - the code is concise
  - they make it easier to work with collections and streams
- However there are some downsides for using Lambda Expressions and functional programming in general:
  - functional programming make the code less readable (and therefore less maintainable)
  - debugging is more difficult

  - video

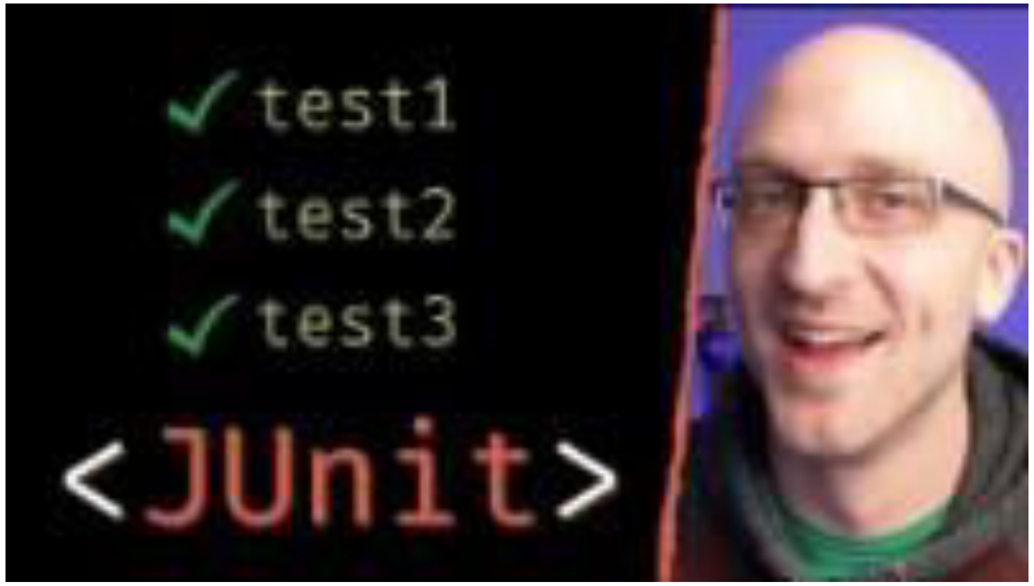
Introduction to Unit Tests

What’s a unit test?

- Automated test
- Fully Controls all the pieces it is testing
- Are isolated
- Are independent of each other
- Runs in memory
- Is consistent
- Fast
- Tests a single concept/class
- Readable
- Maintainable

Unit Testing in IntelliJ

- video



Syntax to remember

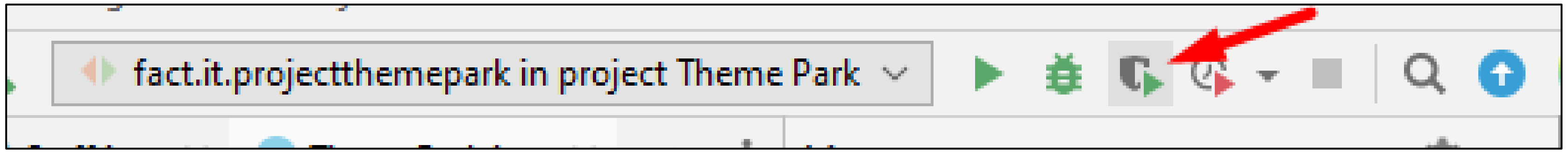
- Interesting links:
  - https://junit.org/junit5/docs/current/api/
  - https://junit.org/junit5/docs/current/api/org.junit.jupiter.api/org/junit/jupiter/api/Assertions.html
- Necessary imports:

```
import org.junit.jupiter.api.Test;
import org.springframework.boot.test.context.SpringBootTest;

import static org.junit.jupiter.api.Assertions.*;
```

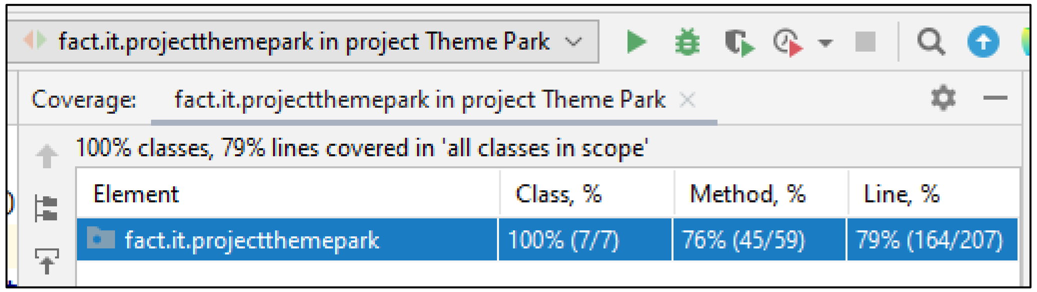
- assertArrayEquals:
  - to compare an expected **array** with an actual value
- assertEquals:
  - to compare an expected **value** with an actual value
- assertNotNull / assertNull:
  - to check whether an actual value is null of not null
- assertTrue / assertFalse:
  - to check whether an actual value is true or false

Testing with coverage information

- After writing the tests, you can check the coverage of your tests with



- Example of result:

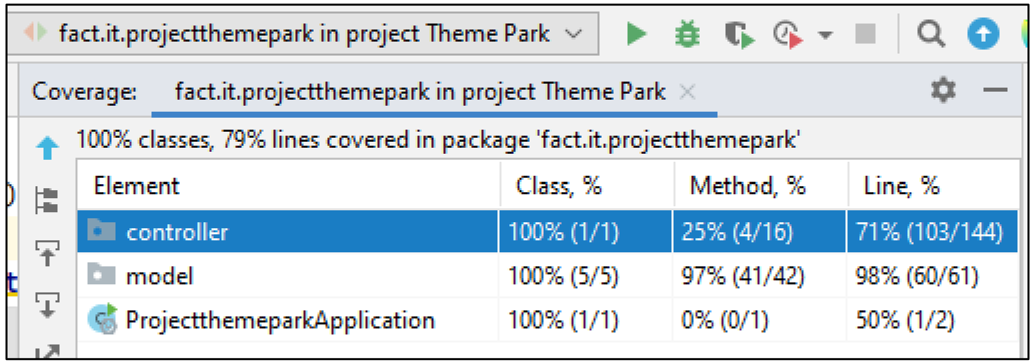


Coverage: fact.it.projectthemepark in project Theme Park

100% classes, 79% lines covered in 'all classes in scope'

Element	Class, %	Method, %	Line, %
fact.it.projectthemepark	100% (7/7)	76% (45/59)	79% (164/207)

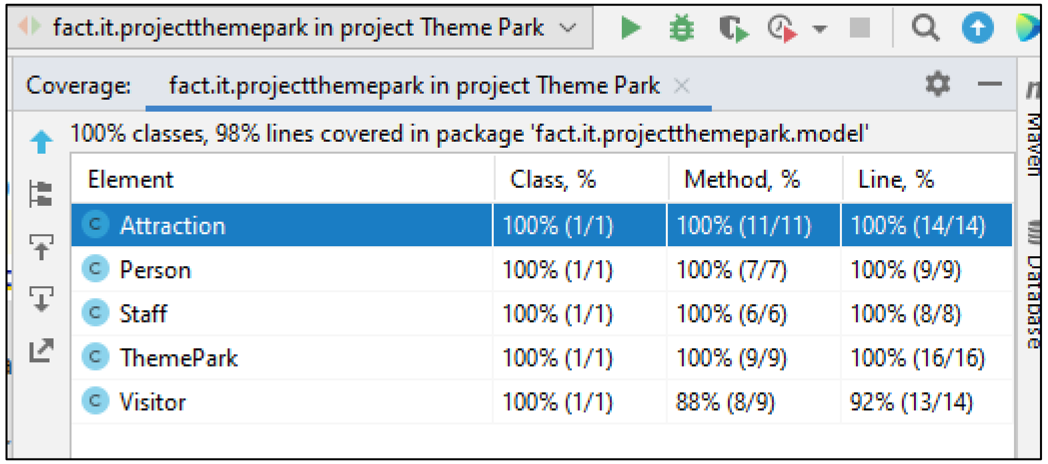
- Opening the folder gives you more insights:



Coverage: fact.it.projectthemepark in project Theme Park

100% classes, 79% lines covered in package 'fact.it.projectthemepark'

Element	Class, %	Method, %	Line, %
controller	100% (1/1)	25% (4/16)	71% (103/144)
model	100% (5/5)	97% (41/42)	98% (60/61)
ProjectthemeparkApplication	100% (1/1)	0% (0/1)	50% (1/2)



Coverage: fact.it.projectthemepark in project Theme Park

100% classes, 98% lines covered in package 'fact.it.projectthemepark.model'

Element	Class, %	Method, %	Line, %
Attraction	100% (1/1)	100% (11/11)	100% (14/14)
Person	100% (1/1)	100% (7/7)	100% (9/9)
Staff	100% (1/1)	100% (6/6)	100% (8/8)
ThemePark	100% (1/1)	100% (9/9)	100% (16/16)
Visitor	100% (1/1)	88% (8/9)	92% (13/14)