**https://www.geeksforgeeks.org/overriding-in-java/?ref=lbp**

**Amigoscode**

[**https://www.interviewbit.com/blog/java-8-features/**](https://www.interviewbit.com/blog/java-8-features/)

**Java8 Features:**

1. **Streams(Java Stream API For Bulk Data Operations On Collections)**

The Stream API is used to handle a collection of items and allows many iterations. A Stream is a collection of items (elements) that enables you to combine multiple techniques to achieve your goals.

Streams allows us to ask what we want from the collection.

We use declarative approach.

We have few methods in this approach,

1. Filter
2. Sort
3. All match
4. Any match
5. None match
6. Max
7. Min
8. Group

Private static List<Person> getPeople(){

Return.List.of(

new Person(name:”James Bond” age: 20, Gender.MALE),

new Person(name:”Bond” age: 33, Gender.FEMALE),

new Person(name:”Alina Bond” age: 57, Gender.MALE),

new Person(name:”Jania Bond” age: 18, Gender.FEMALE),

);

}

In imperative approach,

List<Person> people = getPeople();

List<Person> females = new ArrayList<>();

For(Person person: people){

If(person.getGender().equals(Gender.FEMALE)){

females.add(person);

}

}

Declarative approach,

* **Filter**

Filter takes a Predicate, Predicates can either be true or false.

List<Person> filtered = people.stream()

.filter(person -> person.getGender().equals(Gender.FEMALE))

.collect(Collectors.toList());

* **Sort(in proper order)**

Sort takes a comparator.

List<Person> sorted = people.stream()

.sort(Comparator.comparing(Person :: getAge).reversed())

.collect(Collectors.toList());

* **All match**

Takes a Predicate.

Boolean allMatch = people.stream()

.allMatch(person -> person.getAge() > 5);

* **Any match**

Takes a Predicate.

Boolean anyMatch = people.stream()

.anyMatch(person -> person.getAge() > 5);

* **None match**

Takes a Predicate.

Boolean noneMatch = people.stream()

.noneMatch (person -> person.getName().equals(“Antonia”));

* **Max**

Maxtakes a comparator.

people.stream()

.max(Comparator.comparng(Person :: getAge))

.ifPresent(System.out::println);

* **Min**

Mintakes a comparator.

people.stream()

.min(Comparator.comparng(Person :: getAge))

.ifPresent(System.out::println);

* **Group**

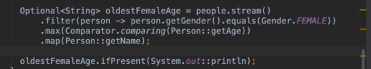
Map<Gender, List<Person> groupByGender = people.stream()

.collect(Collectors.groupingBy(Person::getGender));

groupByGender .forEach((gender, people1) -> {

});

**Common,**



1. **Optionals**

Java Optionals allows you to eliminate NullPointerExceptions. It comes under “java.util” package. It is used to handle NullPointerException in a Java program.

It prevents the application from crashing and terminating unexpectedly. This class has methods for checking the existence of a value for a given variable.

We have few method,

* Optional.empty()
* Optional.of(T value): We are sure that the value is present
* Optional.ofNullable(T value): We are unsure if the value is present or not

We can check if any value is present using isPresent()

We can check if it is empty using isEmpty();

1. **Lambda Expression**

It is very useful in collection library. It helps to iterate, filter and extract data from collection.

The Lambda expression is used to provide the implementation of an interface which has functional interface. It saves a lot of code. In case of lambda expression, we don't need to define the method again for providing the implementation. Here, we just write the implementation code.

Java lambda expression is treated as a function, so compiler does not create .class file.

## Functional Interface

Lambda expression provides implementation of functional interface. An interface which has only one abstract method is called functional interface. Java provides an anotation @FunctionalInterface, which is used to declare an interface as functional interface.

## Why use Lambda Expression

1. To provide the implementation of Functional interface.
2. Less coding.

## Java Lambda Expression Syntax

(argument-list) -> {body}

Java lambda expression is consisted of three components.

**1) Argument-list:** It can be empty or non-empty as well.

**2) Arrow-token:** It is used to link arguments-list and body of expression.

**3) Body:** It contains expressions and statements for lambda expression.

**No Parameter Syntax**

() -> {

//Body of no parameter lambda

}

**One Parameter Syntax**

(p1) -> {

//Body of single parameter lambda

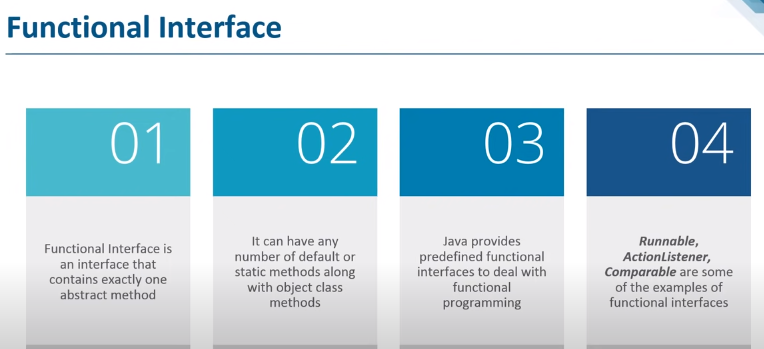
}

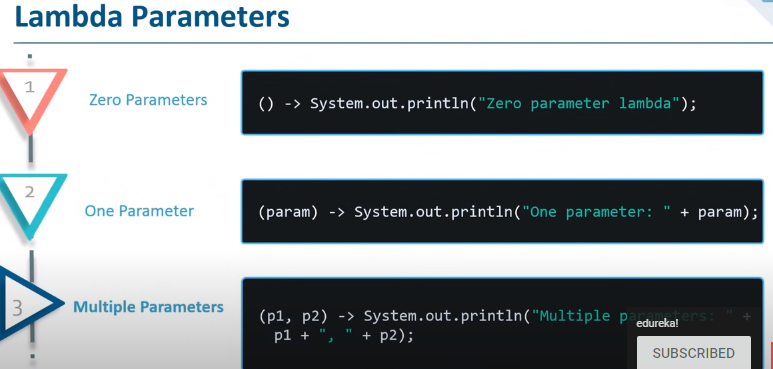
**Two Parameter Syntax**

(p1,p2) -> {

//Body of multiple parameter lambda

}





1. **Collections**

It is an interface

Collection are a group of objects - elements

Some collections allow duplicate elements and others do not.

Some are ordered and some are unordered.

**Set:** No duplicates, No Order maintained

**List:** Maintains Insertion Order, Allows duplicates

Classes cannot directly implement collection interface.

**Implementing classes for List:**

Array cannot be resized

**ArrayList:** Add, contains, get, remove, set, indexOf, addAll, removeAll, lastIndex

* ArrayList can be Re-sizable
* Heterogeneous objects are allowed
* Data Structure: Growable Array

**Adv:** Data Retrieval is faster **disadv:** add/removal of an element at first/middle position needs more shift operations

**LinkedList:** addFirst, addLast, removeFirst, removeLast, getFirst, getLast, pollFirst, pollLast, offerFirst, offerLast, clone, clear

**Adv:** Element Insertion/Deletion is faster **disadv:** Element retrival is not faster

1. **forEach() method in Iterable interface**

In Java 8, the Java.lang interface supports a “forEach” function. Iterable that can iterate over the collection’s items. The Iterable interface has a default method called “forEach.” Collection classes use it to iterate items, which extends the Iterable interface.You may send Lambda Expression as an argument to the “forEach” method, which accepts the Functional Interface as a single parameter.

Example: List<String> subList = new ArrayList<String>();

subList.add("Carrot");

subList.add("Potato");

System.out.println("------------Vegetable List--------------");

subList.forEach(sub -> System.out.println(sub));

1. **Java Date Time API**

Under the package java.time, Java 8 offers a new date-time API. The following are the most prominent classes among them:

Local: Simplified date-time API with no timezone management complexity.

Zoned: specialized date-time API that can handle several time zones.

**Java 11 Features:**

<https://www.geeksforgeeks.org/java-11-features-and-comparison/>

**Java 11 is the first long-term support (LTS) release after Java 8**

**1. New String Methods:  
Java 11 adds a few**[**new methods**](https://www.baeldung.com/java-11-string-api)**to the String class**: isBlank, lines, strip, stripLeading, stripTrailing, and repeat.

isBlank(): This is a boolean method. It just returns true when a string is empty and vice-versa.

lines(): This method is to return a collection of strings that are divided by line terminators.

repeat(n): Result is the concatenated string of original string repeated the number of times in the argument.

stripLeading(): It is used to remove the white space which is in front of the string

strip(): It is used to remove the white spaces which are in front and back of the string

**2. New File Methods**

writeString(): This is to write some content in a file.

readString(): This is used to read the contents of a file.

isSameFile(): This method is used to know whether two paths locate the same file or not.

**3. Pattern Recognizing Methods**

asMatchPredicate(): This method is similar to Java 8 method asPredicate(). Introduced in JDK 11, this method will create a predicate if the pattern matches with the input string.

**4. Collection to an Array**

The java.util.Collection interface contains a new default toArray method which takes an IntFunction argument.

This makes it easier to create an array of the right type from a collection:

**5. Epsilon Garbage Collector**

**6. Local-Variable Syntax for Lambda Parameters**

JDK 11 allows ‘var’ to be used in lambda expressions. This was introduced to be consistent with the local ‘var’ syntax of Java 10.

**7.Running Java Files**

A major change in this version is that **we don't need to compile the Java source files with***javac***explicitly anymore***:*

$ javac HelloWorld.java

$ java HelloWorld

Hello Java 8!

Instead, we can [directly run the file](https://www.baeldung.com/java-single-file-source-code) using the *java*command:

$ java HelloWorld.java

Hello Java 11!