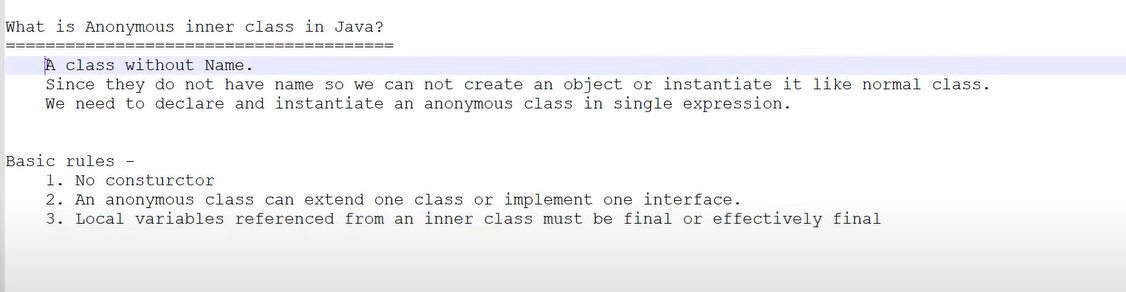
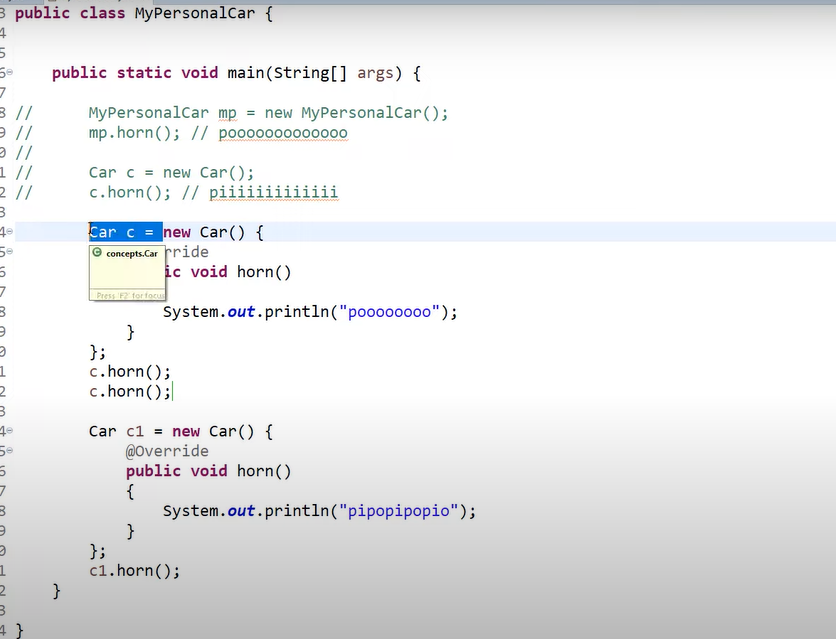
**JAVA**

Graphical user interface, application, Word

Description automatically generated





Graphical user interface

Description automatically generated with low confidence

An interface which has single abstract method. Then that interface is called functional interface.

Note: **Functional interfaces** can have only one abstract class. It can have multiple default and static methods.

**Streams**

A Stream in Java can be defined as a **sequence of elements from a source**. The source of elements here refers to a [Collection](https://howtodoinjava.com/java/collections/into-to-java-collection/) or [Array](https://howtodoinjava.com/series/java-arrays/) that provides data to the Stream.

* Java streams are designed in such a way that most of the stream operations (called **intermediate operations**) return a Stream. This helps to create a chain of stream operations. This is called **stream pipe-lining**.
* Java streams also support the **aggregate or terminal operations** on the elements. The aggregate operations are operations that allow us to express common manipulations on stream elements quickly and clearly, for example, finding the max or min element, finding the first element matching giving criteria, and so on.
* Not that a ***stream maintains the same ordering* of the elements *as the ordering in the stream source***.

The Stream API is used to process collections of objects

A stream is a sequence of objects that supports various methods which can be pipelined to produce the desired result.

The features of Java stream are –

* A stream is not a data structure instead it takes input from the Collections, Arrays or I/O channels.
* Streams don’t change the original data structure, they only provide the result as per the pipelined methods.
* 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.

Collections -> represent group of element/objects as single entity and to store group of objects.

Streams -> Streams are used to process the data from collection.

**Stream Creation**

Streams can be created from different element sources

e.g.

1. collection or array with the help of stream()

2. of() methods

String[] arr = {"a","b","c","d"};

Stream<String> stream=Arrays.*stream*(arr);

stream.map(x-> x.startsWith("d")).forEach(x -> System.***out***.println("Starts with..............."+x));

**Stream.of()**

In the given example, we are creating a stream of a fixed number of integers.

Stream<Integer> stream = Stream.of(1,2,3,4,5,6,7,8,9);

stream.forEach(p -> System.out.println(p));

**Multi-threading With Streams**

Arrays.asList(arr).parallelStream().forEach(

x -> System.***out***.println("Parallel with..............."+x));

**Stream Operations**

1. They are divided into intermediate operations (return Stream<T>) and terminal operations (return a result of definite type). Intermediate operations allow chaining.
2. It's also worth noting that operations on streams don't change the source.

**long** count = list.stream().distinct().count();

So, the distinct() method represents an intermediate operation, which creates a new stream of unique elements of the previous stream. And the count() method is a terminal operation, which returns stream's size.

### ****Iterating****

### Stream API helps to substitute for, for-each, and while loops. It allows concentrating on operation's logic, but not on the iteration over the sequence of elements.

List<String> list =Arrays.*asList*("a","b","c","d");

**boolean** b= list.stream().anyMatch(x -> list.contains("a"));

System.***out***.println("Boolean b is...."+ b);

1. anyMatch
2. allMatch
3. noneMatch

### ****Filtering****

### The filter() method allows us to pick a stream of elements that satisfy a predicate.

ArrayList<String> list = **new** ArrayList<>();

list.add("One");

list.add("OneAndOnly");

list.add("Derek");

list.add("Change");

list.add("factory");

list.add("justBefore");

list.add("Italy");

list.add("Italy");

list.add("Thursday");

list.add("");

list.add("");

list.stream().filter(x -> x.contains("d")).collect(Collectors.*toList*())

.forEach(x -> System.***out***.println(x));

**Mapping**

List<String> uris = **new** **ArrayList**<>(); uris.add("C:\\My.txt"); Stream<Path> stream = uris.stream().map(uri -> Paths.get(uri));

List<Detail> details = **new** **ArrayList**<>(); details.add(**new** **Detail**()); Stream<String> stream = details.stream().flatMap(detail -> detail.getParts().stream());

**Matching**

**boolean** isValid = list.stream().anyMatch(element -> element.contains("h")); // true

**boolean** isValidOne = list.stream().allMatch(element -> element.contains("h")); // false

**boolean** isValidTwo = list.stream().noneMatch(element -> element.contains("h")); // false

**Reduction**

Stream API allows reducing a sequence of elements to some value according to a specified function with the help of the reduce() method of the type Stream. This method takes two parameters: first – start value, second – an accumulator function.

List<Integer> integers = Arrays.asList(1, 1, 1);

**Integer** reduced = integers.stream().reduce(23, (a, b) -> a + b);

Collections are used to store group of objects and streams are used to process the data.

Stream :

1. Filter
2. Map

Filter is used to filter the condition and store the result in the other collection.

**Stream**.

1. Collect
2. Count
3. Sorted
4. Distinct
5. Foreach
6. Min
7. Max

**Non-Terminal methods**

1. Filter()
2. Map()
3. FlatMap()
4. Distinct()
5. Limit()

Note: Non-Terminal methods returns stream of objects.

**Terminal Operations**

1. Collect()
2. Count()
3. Min()
4. Max()
5. Foreach()
6. toArray()

**FlatMap**

FlatMap return group of objects.

FlatMap return multiple values.

Graphical user interface, website

Description automatically generated

Text

Description automatically generated

A screenshot of a computer

Description automatically generated with medium confidence

## Working with Streams

### 7.1 Creating Streams

* concat()
* empty()
* generate()
* iterate()
* of()

### 7.2 Intermediate Operations

* [filter()](https://howtodoinjava.com/java8/java-stream-filter-example/)
* [map()](https://howtodoinjava.com/java8/stream-map-example/)
* [flatMap()](https://howtodoinjava.com/java8/stream-flatmap-example/)
* [distinct()](https://howtodoinjava.com/java8/java-stream-distinct-examples/)
* [sorted()](https://howtodoinjava.com/java8/stream-sorted-method/)
* [peek()](https://howtodoinjava.com/java8/java-stream-peek-example/)
* [limit()](https://howtodoinjava.com/java8/java-stream-limit-method-example/)
* [skip()](https://howtodoinjava.com/java8/stream-skip-example/)

### 7.3. Terminal Operations

* [forEach()](https://howtodoinjava.com/java8/java-stream-foreach/)
* [forEachOrdered()](https://howtodoinjava.com/java8/java-stream-foreachordered/)
* [toArray()](https://howtodoinjava.com/java8/convert-stream-to-array/)
* reduce()
* collect()
* [min()](https://howtodoinjava.com/java8/java-stream-min/)
* [max()](https://howtodoinjava.com/java8/java-stream-max/)
* [count()](https://howtodoinjava.com/java8/stream-count-elements-example/)
* [anyMatch()](https://howtodoinjava.com/java8/stream-anymatch-example/)
* [allMatch()](https://howtodoinjava.com/java8/stream-allmatch-example/)
* [noneMatch()](https://howtodoinjava.com/java8/stream-nonematch-example/)
* [findFirst()](https://howtodoinjava.com/java8/stream-findfirst-findany/)
* [findAny()](https://howtodoinjava.com/java8/stream-findfirst-findany/)

Stream<Integer> randomNumbers = Stream.generate(() -> (**new** Random()).nextInt(100));

randomNumbers.limit(20).forEach(System.out::println);

**LAMDA EXPRESSIONS From 1.8**

* 1. Java doesn’t support functional programming features.
  2. So, it bought lamda expressions in java
  3. To bring code optimization in java
  4. Where code is written in functions and variables.

1. Lamda Expression is anonymous function.
2. Anonymous function ->
   * 1. It doesn’t have name
     2. doesn’t have return type
     3. no access modifier.
3. In Lamda expression we don’t need to specify the data type of the variable.
4. Java compiler identifies type of the variable based on the context.

**public** **void** m()

{

System.out.println("Lamda expression");

}

**() -> System.out.println("Lamda expression");**

**public** **void** m(**int** a, **int** b)

{

System.out.println("Lamda expression");

}

**(a,b) -> System.out.println(a+b);**

Text, letter

Description automatically generated

Text, letter

Description automatically generated

**Functional Interface**

To invoke lamda expression, we need to use functional interface.

**Functional Interface :** Which contains **SINGLE ABSTRACT METHOD**.

From Java 7. In Interface we can create

* 1. Abstract methods.
  2. Default methods.
  3. Static methods.

As far as Functional Interface is concerned

1. It should contain SINGLE Abstract Method.
2. Any number of DEFAULT method.
3. Any number of STATIC method.

**Default Functional Interface**

1. Runnable -> run()
2. Callable -> call()
3. Comparable -> compareTo()
4. ActionListener -> actionPerformed()

Note : To invoke lamda expressions, we need to use functional interface.

Graphical user interface, text, application

Description automatically generated

Note : Lamda expressions are associated with the functional interfaces.

Lamda expressions are not associated with other interfaces.

For abstract method in functional interface, we can use the lamda expression.

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Note : We can write lamda expression only for functional interfaces.

In java we have predefined interfaces.

**Pre-defined Functional interfaces.**

By using pre-defined functional interfaces, we can use lamda expression.

1. Predicate
2. Function **Java.util.function**
3. Consumer
4. Supplier

The above predefined functional interfaces have only one abstract method.

**Predicate – Functional Interface.**

1. It has only single abstract method. i.e test() and it takes single argument, and it returns boolean value.

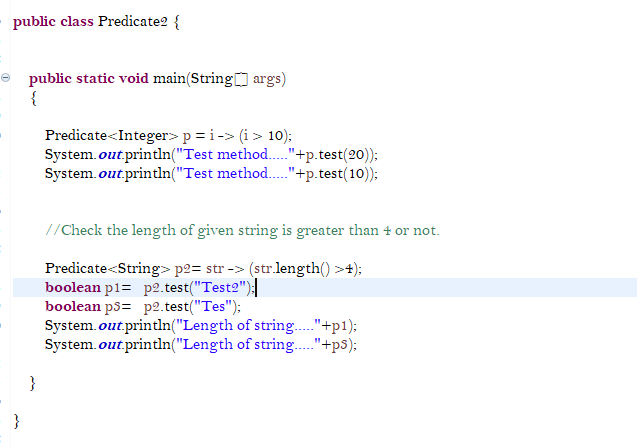
We use predicate interfaces to check conditions in lamda expressions.

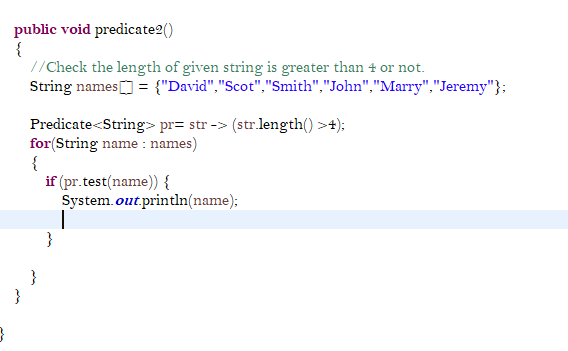
Interface Predicate<T>

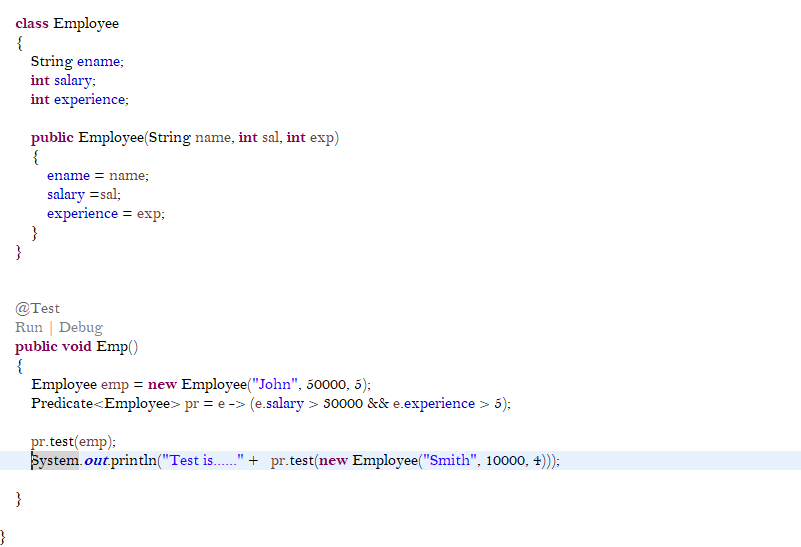
{

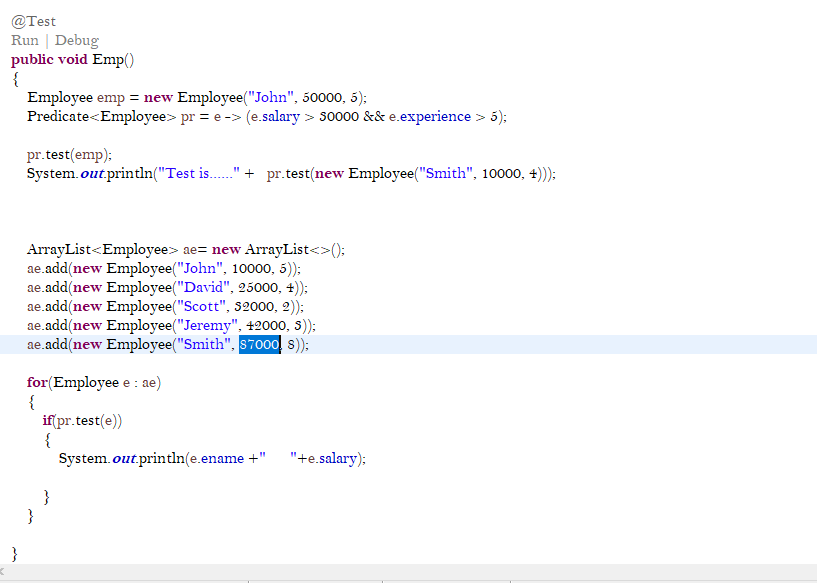
public abstract boolean test(T);

}









AND Operator in predicate

Graphical user interface, text, application

Description automatically generated

OR operator in Predicator

Graphical user interface, text, application, email

Description automatically generated

Negate

Graphical user interface, text, application, email

Description automatically generated

Function Interface --

Function interface has Method ---- apply()

1. Apply()
2. Any type as parameter
3. Return single value and it can be any value.

\* **@param** t the function argument

\* **@return** the function result

Interface Function<T,R>

{

R apply(T);

}

**public** **void** Function()

{

Function<Integer,Integer> f = n -> n\*n;

f.apply(5);

//String length

//String ----> length ---> Int

Function<String, Integer> fn = s -> s.length();

System.***out***.println(fn.apply("Welcome to Streams"));

ArrayList<Employee> ae= **new** ArrayList<Employee>();

ae.add(**new** Employee("John", 10000));

ae.add(**new** Employee("David", 25000));

ae.add(**new** Employee("Scott", 32000));

ae.add(**new** Employee("Jeremy", 42000));

ae.add(**new** Employee("Smith", 87000));

Function<Employee, Integer> fn2 =

e ->

{

**int** sal = e.salary;

**if** (sal >= 30000)

{

**return** (sal\* 10/100);

}**else**

{

**return** (sal\* 40/100);

}

};

Predicate<Integer> p = b -> b > 50000;

**for**(Employee ep :ae)

{

**int** bonus=fn2.apply(ep);

**if** (p.test(bonus))

{

System.***out***.println(ep.ename+" "+ ep.salary);

System.***out***.println("Bonus is....."+ bonus);

}

}

}

Text, letter

Description automatically generated