Java8.

Functional Interface:

An Interface that contains only one abstract method is known as functional interface. It can have any number of default and static methods. It can also declare methods of object class.

Functional interfaces are also known as Single Abstract Method Interfaces (SAM Interfaces).

java.util.function package interfaces are categorized into 4:

1. Predicate

2. Consumer

3. Supplier

4. Function

++++++++++++++++++++++++++++++++++++++++++++++

public interface Consumer<T> {

public void accept(T t);

}

It represents a function which takes in one argument and produces a result. However these kind of functions don’t return any value.

Hence this functional interface which takes in one generic namely:-

* + **T**: denotes the type of the input argument to the operation

The lambda expression assigned to an object of Consumer type is used to define its **accept()** which eventually applies the given operation on its argument. Consumers are useful when it not needed to return any value as they are expected to operate via side-effects.

+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

public interface Predicate<T> {

public boolean test(T t);

}

It is a functional interface which represents a predicate (boolean-valued function) of one argument. It is defined in the java.util.function package and contains test() a functional method.

Java Predicate Interface Methods

|  |  |
| --- | --- |
| **Methods** | **Description** |
| boolean test(T t) | It evaluates this predicate on the given argument. |
| default Predicate<T> and(Predicate<? super T> other) | It returns a composed predicate that represents a short-circuiting logical AND of this predicate and another. When evaluating the composed predicate, if this predicate is false, then the other predicate is not evaluated. |
| default Predicate<T> negate() | It returns a predicate that represents the logical negation of this predicate. |
| default Predicate<T> or(Predicate<? super T> other) | It returns a composed predicate that represents a short-circuiting logical OR of this predicate and another. When evaluating the composed predicate, if this predicate is true, then the other predicate is not evaluated. |
| static <T> Predicate<T> isEqual(Object targetRef) | It returns a predicate that tests if two arguments are equal according to Objects.equals(Object, Object). |

*++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*

public interface Supplier<T> {

public T get();

}

It represents a function which does not take in any argument but produces a value of type T.

Hence this functional interface takes in only one generic namely:-

* + **T**: denotes the type of the result

The lambda expression assigned to an object of Supplier type is used to define its **get()** which eventually produces a value. Suppliers are useful when we don’t need to supply any value and obtain a result at the same time.

The Supplier interface consists of only one function:

**1. get()**

This method does not take in any argument but produces a value of type T.

**Syntax:**

T get()

**Returns:** This method returns a **value** of type T.

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

public interface Function<T, R> {

public R apply(T t);

}

It represents a function which takes in one argument and produces a result. Hence this functional interface takes in 2 generics namely as follows:

* + **T**: denotes the type of the input argument
  + **R**: denotes the return type of the function

*The lambda expression assigned to an object of Function type is used to define its****apply()****which eventually applies the given function on the argument.*

**Methods in Function Interface**

The Function interface consists of the following 4 methods as listed which are later discussed as follows:

* + apply()
  + andThen()
  + compose()
  + identity()

+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

Lambda Expression:

Lambda expressions are used primarily to define inline implementation of a functional interface, i.e., an interface with a single method only.

Lambda expression eliminates the need of anonymous class and gives a very simple yet powerful functional programming capability to Java.

A lambda expression is a concise representation of an anonymous function that can be passed around, it does not have a name, it has list of parameters, a body, a return type and possibly a list of exceptions that can be thrown.

# Anonymous => Its anonymous because it doesn't have an explicit name.

# Function => We say function because a lambda isn't associated with a particular class like a method is. But like a method, a lambda expression has list of parameters, return type and a list of exceptions that can be thrown..

# Passed around => A lamda expression can be passed as an argument to a method or stored in a variable.

# Concise => You don't need to write a boilerplate code like we do for anonymous classes.

*Stream API:*

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.

Q. How to build Streams?

==>

1. From collection object

List<Dish> menu = new ArrayList<>();

Stream<Dish> stream = menu.stream();

2. Empty stream

Stream stream = Stream.empty();

3. of() method that accepts single parameter.

Stream<String> stream = Stream.of("Anna");

4. of() method that accepts multiple parameters.

Stream<String> stream = Stream.of("Anna", "Alex", "Bob", "Peter");

5. A stream on the lines of text files.

Stream<String> stream = Files.lines(path);

6. Create a stream from array.

int arr[] = { 10, 20, 30, 40, 10, 20, 60, 80, 90, 30};

IntStream stream = Arrays.stream(arr);

stream.distinct()

.forEach(n -> System.out.println(n));

7. Create a stream with infinite data.

**Intermediate Operations:**

* + **map:**The map method is used to returns a stream consisting of the results of applying the given function to the elements of this stream.  
    List number = Arrays.asList(2,3,4,5);  
    List square = number.stream().map(x->x\*x).collect(Collectors.toList());
  + **filter:** The filter method is used to select elements as per the Predicate passed as argument.  
    List names = Arrays.asList("Reflection","Collection","Stream");  
    List result = names.stream().filter(s->s.startsWith("S")).collect(Collectors.toList());
  + **sorted:** The sorted method is used to sort the stream.  
    List names = Arrays.asList("Reflection","Collection","Stream");  
    List result = names.stream().sorted().collect(Collectors.toList());

**Terminal Operations:**

* + **collect:** The collect method is used t o return the result of the intermediate operations performed on the stream.  
    List number = Arrays.asList(2,3,4,5,3);  
    Set square = number.stream().map(x->x\*x).collect(Collectors.toSet());
  + **forEach:** The forEach method is used to iterate through every element of the stream.  
    List number = Arrays.asList(2,3,4,5);  
    number.stream().map(x->x\*x).forEach(y->System.out.println(y));
  + **reduce:** The reduce method is used to reduce the elements of a stream to a single value.  
    The reduce method takes a BinaryOperator as a parameter.  
    List number = Arrays.asList(2,3,4,5);  
    int even = number.stream().filter(x->x%2==0).reduce(0,(ans,i)-> ans+i);  
    Here ans variable is assigned 0 as the initial value and i is added to it .

|  |  |
| --- | --- |
| **STREAMS** | **COLLECTIONS** |
| It doesn’t store data, it operates on the source data structure i.e collection. | It stores/holds all the data that the data structure currently has in a particular data structure like Set, List or Map, |
| They use functional interfaces like lambda which makes it a good fit for programming language. | They don’t use functional interfaces. |
| Java Streams are consumable i.e; to traverse the stream, it needs to be created every time. | They are non-consumable i.e; can be traversable multiple times without creating it again. |
| Java streams support both sequential and parallel processing. | It supports parallel processing and parallel processing can be very helpful in achieving high performance. |
| All the Java stream API interfaces and classes are in j**ava.util.stream** package. | Specific classes for primitive types such as **IntStream**, **LongStream**, and **DoubleStream** are used in collections since primitive data types such as int, long in the collections using auto-boxing and these operations could take a lot of time. |
| Streams are not modifiable i.e one can’t add or remove elements from streams. | These are modifiable i.e one can easily add to or remove elements from collections. |
| Streams are iterated internally by just mentioning the operations. | Collections are iterated externally using loops |

***Sequential Stream:***

Sequential Streams are non-parallel streams that use a single thread to process the pipelining. Any stream operation without explicitly specified as parallel is treated as a sequential stream. Sequential stream’s objects are pipelined in a single stream on the same processing system hence it never takes the advantage of the multi-core system even though the underlying system supports parallel execution. Sequential stream performs operation one by one.

**stream()** method returns a sequential stream in Java.

*Parallel Stream:*

It is a very useful feature of Java to use parallel processing, even if the whole program may not be parallelized. Parallel stream leverage multi-core processors, which increases its performance. Using parallel streams, our code gets divide into multiple streams which can be executed parallelly on separate cores of the system and the final result is shown as the combination of all the individual core’s outcomes. It is always not necessary that the whole program be parallelized, but at least some parts should be parallelized which handles the stream. The order of execution is not under our control and can give us unpredictably unordered results and like any other parallel programming, they are complex and error-prone.

The Java stream library provides a couple of ways to do it. easily, and in a reliable manner.

* + One of the simple ways to obtain a parallel stream is by invoking the [parallelStream()](https://www.geeksforgeeks.org/what-is-java-parallel-streams/) method of **Collection**interface.
  + Another way is to invoke the[parallel()](https://www.geeksforgeeks.org/what-is-java-parallel-streams/)method of **BaseStream** interface on a sequential stream.

It is important to ensure that the result of the parallel stream is the same as is obtained through the sequential stream, so the parallel streams must be stateless, non-interfering, and associative.

**Note:** If we want to make each element in the parallel stream to be ordered, we can use the [forEachOrdered()](https://www.geeksforgeeks.org/stream-foreachordered-method-java-examples/) method, instead of the [forEach()](https://www.geeksforgeeks.org/stream-foreach-method-java-examples/) method.

|  |  |
| --- | --- |
| Sequential Stream | Parallel Stream |
| Runs on a single-core of the computer | Utilize the multiple cores of the computer. |
| Performance is poor | The performance is high. |
| Order is maintained | Doesn’t care about the order, |
| Only a single iteration at a time just like the for-loop. | Operates multiple iterations simultaneously in different available cores. |
| Each iteration waits for currently running one to finish, | Waits only if no cores are free or available at a given time, |
| More reliable and less error, | Less reliable and error-prone. |
| Platform independent, | Platform dependent |

**Conclusion**

The stream APIs have been a part of Java for a long time for its intriguing feature. It is also very much popular for parallel processing capability and improved performance.  In this era literally, every modern machine is multi-core so to this core efficiently we should use parallel streams however parallel programming design is complex. So it’s completely up to the programmer whether he wants to use parallel streams or sequential streams based on the requirements.

Optional Class:

Java 8 has introduced a new class Optional in **java.util package**. It can help in writing a neat code without using too many null checks. By using Optional, we can specify alternate values to return or alternate code to run. This makes the code more readable because the facts which were hidden are now visible to the developer.

## Creating Optional Object:

1. Empty Optional

Optional<Car> car = Optional.empty();

2. Optional From non-null values

Optional<Car> car = Optional.of(car);

// If car object is null, NullPointerException will be thrown immediately once you try to acess properties of Car.

3. Optional from null.

Optional<Car> car = Optional.ofNullable(car);

// If car object is null, then it will return empty Optional.