

What are the new features introduced in java 8?

- Lambda Expressions
- Functional Interface
- Default and static methods
- Stream API
- Optional class
- forEach method

Created By - Shital Jadhav

Java Lambda Expressions and Functional Interface

Created By - Shital Jadhav

Lambda Expressions

- Lambda Expression is an anonymous function. It's a function without name and does not belongs to any classes.
- It is mainly used to provide implementation of Functional Interface.
- Syntax. (List of arguments) -> { implementation/body } ;

Functional Interface

- The interface which contains only one abstract method is known as functional interface. It may contain any number of default and static method.
- We can use `@FunctionalInterface` annotation to make a interface as functional interface .
- Predefined FI examples are `Runnable` , `Comparable` etc.

Types OF Functional Interfaces

- `Function<T, R>`
- `Consumer<T>`
- `Predicate<T>`
- `Supplier<T>`

Created By - Shital Jadhav

Function<T, R>

- Function interface is available in java.util.function package.
- Represents a function that **accepts one argument** and **produces a result**.
- Type Parameters

T - the type of the input to the function

R - the type of the result of the function

- **R apply(T t)**
- One of the common use cases of this interface is Stream.map method

Example of Function Interface

```
public class FunctionDemo {  
    public static void main(String[] args) {  
        Function<Integer,Integer> function=new FunctionImpl();  
        System.out.println(function.apply(8));  
    }  
}  
1 usage  
class FunctionImpl implements Function<Integer,Integer>{  
    @Override  
    public Integer apply(Integer num) {  
        return num*num;  
    }  
}
```

Without Lambda Expression

```
public class FunctionDemo {  
    public static void main(String[] args) {  
        Function<Integer,Integer> function=(num)-> num*num;  
        System.out.println(function.apply(8));  
    }  
}
```

With Lambda Expression

Consumer<T>

- Represents an operation that **accepts a single input argument** and **returns no result**.
 1. **void accept(T t);**
 2. **default Consumer<T> andThen(Consumer<? super T> after);**
- A variety of methods in the Java Stream API take the functional Consumer interface as an argument, including methods such as collect, forEach .

Example of Consumer Interface

```
public class ConsumerDemo {  
    public static void main(String[] args) {  
        Consumer<String> consumer=new ConsumerImpl();  
        consumer.accept(t:"sj programming solutions");  
    }  
}  
1 usage  
class ConsumerImpl implements Consumer<String>{  
    @Override  
    public void accept(String s) {  
        System.out.println(s.toUpperCase());  
    }  
}
```

Without Lambda Expression

```
public class ConsumerDemo {  
    public static void main(String[] args){  
        Consumer<String> consumer=(s)-> System.out.println(s.toUpperCase());  
        consumer.accept(t:"sj programming solutions");  
    }  
}
```

With Lambda Expression

Predicate<T>

- A Predicate interface represents a boolean-valued-function of an argument.
- This is mainly used to filter data from a Java Stream.
- The filter method of a stream accepts a predicate to filter the data and return a new stream satisfying the predicate.
- A predicate has a test() method which **accepts an argument** and **returns a boolean value**.
- **boolean test(T t)**

Example of Predicate Interface

```
public class PredicateDemo {  
    public static void main(String[] args){  
        Predicate<String> predicate=new PredicateImpl();  
        System.out.println(predicate.test(t: "programming"));  
    }  
}  
1 usage  
class PredicateImpl implements Predicate<String>{  
    @Override  
    public boolean test(String s) {  
        return s.length()>5;  
    }  
}
```

Without Lambda Expression

```
public class PredicateDemo {  
    public static void main(String[] args){  
        Predicate<String> predicate=(s)->s.length()>5;  
        System.out.println(predicate.test(t: "programming"));  
    }  
}
```

With Lambda Expression

Supplier<T>

- It represents a function which does not take in any argument but produces a value of type T.
- Suppliers are useful when we don't need to supply any value and obtain a result at the same time.
- `T get()` : does not take in any argument but produces a value of type T.

Example of Supplier Interface

```
public class SupplierDemo {  
    public static void main(String[] args){  
        Supplier<Double> supplier=new SupplierImpl();  
        System.out.println(supplier.get());  
    }  
}  
1 usage  
class SupplierImpl implements Supplier<Double>{  
    @Override  
    public Double get() {  
        return Math.random();  
    }  
}
```

Without Lambda Expression

```
public class SupplierDemo {  
    public static void main(String[] args){  
        Supplier<Double> supplier=()->Math.random();  
        System.out.println(supplier.get());  
    }  
}
```

With Lambda Expression

Default Methods in Java 8

- Default methods enable you to add new functionality to the interfaces of your libraries and ensure binary compatibility with code written for older versions of those interfaces.
- The default methods are fully implemented methods in an interface, and they are declared by using the keyword default.
- Because the default methods have some default implementation, they help extend the interfaces without breaking the existing code.

What is Optional Class ?

- A container object which may or may not contain a non-null value.
- The **Optional** is a wrapper class that makes a field optional which **means it may or may not have values**. It improves the readability .
- By using Optional, you can specify alternative values to return when something is null.
- EX. if you have an Employee Object and it has yet to assign a department, **instead of returning null, you can return a default department**. Earlier, there was no option to specify such default value in Java code but from Java 8 onwards, Optional can be used for that.

How to create Optional Class ?

1. **Optional.empty()** :

```
String name="Tom";
```

```
Optional<Object> emptyOptional = Optional.empty();
```

2. **Optional.of(T value)** :

```
Optional<String> nameOptional = Optional.of(name);
```

3. **Optional.ofNullable(T value)** :

```
Optional<String> stringOptional = Optional.ofNullable(name);
```


Methods From Optional Class

- `get()`
- `isPresent()`
- `ifPresentOrElse(Consumer<? Super T> action , Runnable emptyAction)`
- `orElse(T other)`
- `orElseThrows()`

Created By - Chital Jadhav

Stream API in java 8

- Stream API is used to process collections of objects.
- Stream is a pipeline consists of a source zero or more intermediate operations (which transform a stream into another stream, such as `filter(Predicate)`), and a terminal operation (which produces a result or side-effect, such as `count()` or `forEach(Consumer)`).

What are intermediate and terminal Operations in stream ?

- The operations which return another stream as a result are called intermediate operations.
- Ex. `map()`, `filter()`, `distinct()`, `sorted()`, `limit()`, `skip()`
- The operations which return non-stream values like primitive or object or collection or return nothing are called terminal operations.
- Ex.
- `forEach()`, `toArray()`, `reduce()`, `collect()`, `min()`, `max()`, `count()`, `anyMatch()`, `allMatch()`, `noneMatch()`, `findFirst()`, `findAny()`

