JAVA 8 FEATURES

Presented By-

Krati Tamrakar

Java Programming provides following features in version 8 :-

1. Functional Interfaces and Lambda Expressions
2. forEach() method
3. Method References
4. Java Stream API
5. Default and Static methods
6. Java Date and Time API
7. Optional Class

Lambda Expressions

- An easy expressive way to define anonymous functions and expressions.
- It is used to provide implementation of functional interfaces, hence saves a lot of code.
- You don't need to define the method again for providing implementation.

Format: <Argument List> **->** <Body>

Example-

```
        Func obj = message->
System.out.println("Hello " + message);
```

Functional Interfaces

- An interface that contains only one abstract method is known as a functional interface.

- These interfaces can be easily represented with lambda expressions.

- You can have **n** number of default and static methods inside a functional interface.

- Java provides an annotation *@FunctionalInterface*, which is used to declare an interface as functional interface.

Example

```
@FunctionalInterface
public interface SimpleFuncInterface {
    public void doWork();
}
```

There are 4 types of functional interfaces introduced in Java 8.

- **Predicate:-** A predicate functional interface of Java is a type of function that accepts a single value as an argument and returns a boolean (True/ False) value.

Syntax-

```
public interface Predicate<T>
{       boolean test(T input);
                }
```

- **Consumer:-** The consumer interface of the functional interface is the one that accepts only one argument and no return value.

Syntax-

```
public interface Consumer <T>
        {  void accept(T t);
       }
```

- **Function :-** A function is a type of functional interface in Java that receives only a single argument and returns a value after the required processing.

Syntax-

```
public interface Function<T, R>
   {
     R apply(T t);
                        }
```

- **Supplier :-** The Supplier functional interface is also a type of functional interface that does not take any input or argument and yet returns a single output.

Syntax-

```
 public interface Supplier<T>
{
  T.get();
                }
```

for-each() Method

The ***forEach()*** method in Java is a utility function to iterate over a *Collection* (list, set or map) or stream elements.

- Easy way to perform a given Consumer action on each item.
- You can supply a lambda expression or method reference to loop over Stream elements.

Syntax - `Void forEach(Consumer< super T> action)`

Example

```
Employees.forEach(e -> e.setSalary(e.getSalary() * 11/10))
```

Give all employees a 10% raise

Method References

- A Method Reference is the shorthand syntax for a lambda expression that contains just one method call.

- It is used to refer method of functional interface.

Format: <Class or Instance>**::**<Method>

If I want to change given String into lower case then,

Using lambda Expression :- s->s.toLowerCase()

Using method reference :- String :: toLowerCase()

Types of Method References

There are mainly type of method references that are as follows:

Type 1: Reference to a static method

Used to refer to static method defined in the class.

Syntax - Class::staticMethodName
Example – Math:: max

Type 2: Reference to an Instance method

Used to refer to instance method defined in the class.

Syntax - Object::instanceMethodName
Example – System.out ::println

Type 3: Reference to a Constructor

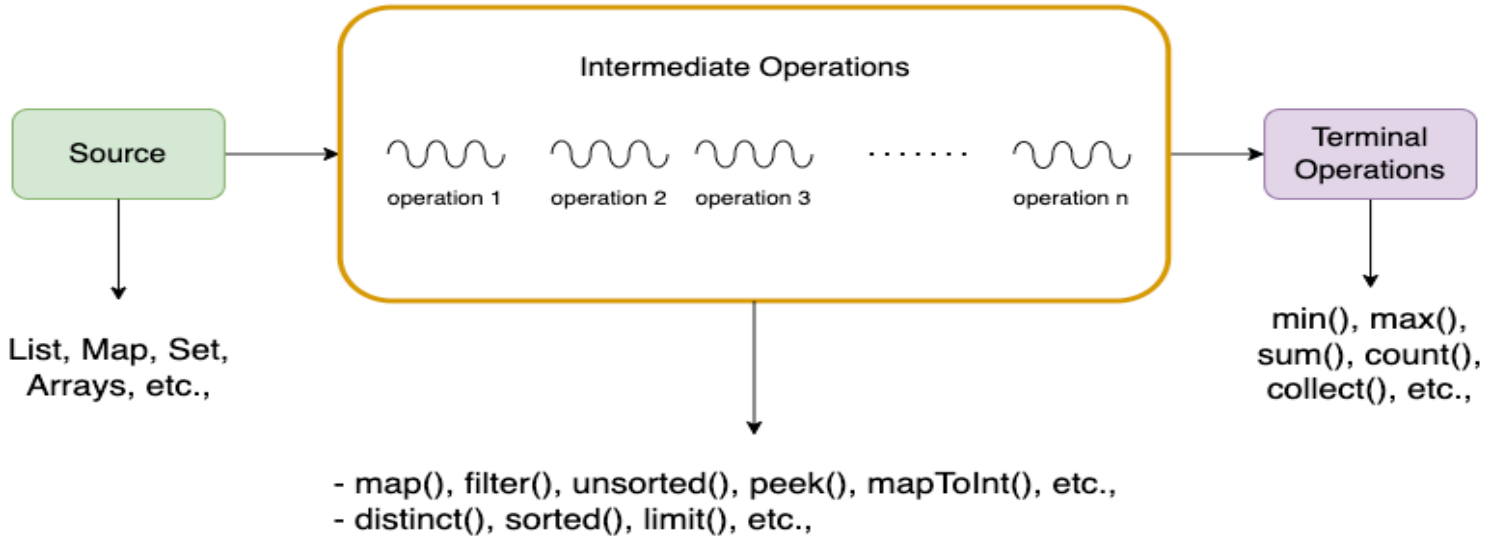
Used to refer to constructor defined in the class by using new keyword.

Syntax - ClassName::new

Example – ArrayList::new

Stream API

- A Stream is processed numerous operations on the elements.
- A Stream starts with a source data structure.
- Intermediate methods are performed on the Stream elements. These methods produce Streams and are not processed until the terminal method is called.



Common Stream API Methods Used

1. Stream<T> filter(Predicate) - Produces a new Stream that contains only the elements of the original Stream that pass a given test.

Example -

employees.filter(e -> e.getSalary() > 100000)

Produce a Stream of Employees with a high salary

2. Stream<T> map(Function) - Produces a new Stream that is the result of applying a Function to each element of original Stream.

Example -

Id.map(EmployeeUtils::findEmployeeById)

Create a new Stream of Employee id

3. Object[] toArray(Supplier) -

Reads the Stream of elements into an array.

Example-

```
Employee[] empArray = employees.toArray(Employee[]::new);
```

Create an array of Employees out of the Stream of Employees

4. List<T> collect(Collectors.toList()) -

Reads the Stream of elements into a List or any other collection.

Example-

```
List<Employee> empList =employees.collect(Collectors.toList());
```

Create a List of Employees out of the Stream of Employees

5. T reduce(T identity, BinaryOperator) –

You start with a seed (identity) value, then combine this value with the first Entry in the Stream, combine the second entry of the Stream, etc.

Example-

```
Nums.stream().reduce(1, (n1,n2) -> n1*n2)
```

Calculate the product of numbers

6. Stream<T> limit(long maxSize) –

Limit(n) returns a stream of the first n elements.

Example-

```
someLongStream.limit(10)
```

First 10 elements

7. Stream<T> distinct() -

Returns a stream consisting of the distinct elements of this stream.

Example-

```
List<Integer> e_id=Arrays.asList(9, 10, 9, 10, 9, 10);

List<Employee> emp =id.stream(). distinct() .collect(Collectors.toList());
```

Get a list of distinct Employees

8. long count() –

Returns the count of elements in the Stream.

Example-

```
employeeStream.filter(somePredicate).count()
```

How many Employees match the criteria?

Default Methods

Before Java 8 –

An Interface doesn't have body.

The only way to add functionality to Interfaces was to declare additional methods which would be implemented in classes that implement the interface.

It is impossible to add methods to an interface without breaking the existing implementation.

Solution –

Java 8 provides a facility to create methods interfaces with their full implementation inside the interface.
You can override default method also to provide more specific implementation for the method.
Allows the addition of functionality to interfaces while preserving backward compatibility.

Static Methods

These methods are similar to default methods but we can't override them in implementation classes.

They are also used for providing utility methods such as null check and collection sorting etc.

Implementation:-

```
 public interface MyData {

default void print(String str) {
        if (!isNull(str))
                System.out.println("MyData Print::" + str);
}

static boolean isNull(String str) {
        System.out.println("Interface Null Check");

        return str == null ? true : "".equals(str) ? true : false;
}))
```

Date/Time API

Java provide the date and time functionality with the help of two packages:-
java.util.Date and java.util.Calendar.

Then Java 8 introduced a new package called java.time to address the shortcomings of the older packages for Date and Time.

some of the core classes of the new Java 8 project that are part of the *java.time* package, such as *LocalDate*, *LocalTime*, *LocalDateTime*, *ZonedDateTime*, *Period*, *Duration* and their supported APIs.

Optional<T> Class

- A container object which may or may not contain a non-null value.
- This class has various utility methods to facilitate code to handle values as ‘available’ or ‘not available’ instead of checking null values.
- Common methods

➢ **isPresent()** – returns true if value is present otherwise false.

➢ **Get()** – returns value if present otherwise throws NoSuchElementException.

➢ **orElse(T other)** – returns value if present, or otherwise return other.

➢ **ifPresent(Consumer)** –if value is present invoked the specified consumer with value otherwise do nothing.

Syntax –

public final class Optional<T> extends Object

Thank You !!