**DAY 1**

Q1.Features of Java 8 with egs:

1. Lambda Expression

A lambda expression is an anonymous function. A function that doesn’t have a name and doesn’t belong to any class.

Syntax: (parameter\_list) -> {function\_body}

For example, the lambda expression (x, y) -> x + y specifies that lambda expression takes two arguments x and y and returns the sum of these.

Lambda expression **only has body and parameter list**.  
1. **No** name – function is anonymous so we don’t care about the name  
2. Parameter list  
3. Body – This is the main part of the function.  
4. **No** return type – The java 8 compiler is able to infer the return type by checking the code.

To use lambda expression, you need to either create your own functional interface or use the pre defined functional interface provided by Java. An interface with **only single abstract method** is called functional interface(or Single Abstract method interface), for example: Runnable, callable, ActionListener etc.

@FunctionalInterface

interface MyFunctionalInterface {

//A method with single parameter

public int incrementByFive(int a);

}

public class Example {

public static void main(String args[]) {

// lambda expression with single parameter num

MyFunctionalInterface f = (num) -> num+5;

System.out.println(f.incrementByFive(22));

}

}

2.Functional Interfaces

An interface with **only single abstract method** is called functional interface. You can either use the predefined functional interface provided by Java or create your own functional interface and use it. To use [lambda expression in Java](https://beginnersbook.com/2017/10/java-lambda-expressions-tutorial-with-examples/), you need to either create your own functional interface or use the pre defined functional interface provided by Java.

@FunctionalInterface

interface MyFunctionalInterface {

public int addMethod(int a, int b);

}

public class BeginnersBookClass {

public static void main(String args[]) {

// lambda expression

MyFunctionalInterface sum = (a, b) -> a + b;

System.out.println("Result: "+sum.addMethod(12, 100));

}

}

3.Default and static methods in interfaces

Java 8 allows the interfaces to have default and static methods. The reason we have default methods in interfaces is to allow the developers to add new methods to the interfaces without affecting the classes that implements these interfaces.

**Static methods** in interfaces are similar to the default methods except that we cannot override these methods in the classes that implements these interfaces.

interface MyInterface{

/\* This is a default method so we need not

\* to implement this method in the implementation

\* classes

\*/

default void newMethod(){

System.out.println("Newly added default method");

}

/\* This is a static method. Static method in interface is

\* similar to default method except that we cannot override

\* them in the implementation classes.

\* Similar to default methods, we need to implement these methods

\* in implementation classes so we can safely add them to the

\* existing interfaces.

\*/

static void anotherNewMethod(){

System.out.println("Newly added static method");

}

/\* Already existing public and abstract method

\* We must need to implement this method in

\* implementation classes.

\*/

void existingMethod(String str);

}

public class Example implements MyInterface{

// implementing abstract method

public void existingMethod(String str){

System.out.println("String is: "+str);

}

public static void main(String[] args) {

Example obj = new Example();

//calling the default method of interface

obj.newMethod();

//calling the static method of interface

MyInterface.anotherNewMethod();

//calling the abstract method of interface

obj.existingMethod("Java 8 is easy to learn");

}

}

3.Streams

All the classes and interfaces of this API is in the java.util.stream package. By using streams we can perform various aggregate operations on the data returned from collections, arrays, Input/Output operations.

1. Create a stream

2. Perform **intermediate operations** on the initial stream to transform it into another stream and so on on further intermediate operations.

3. Perform **terminal operation** on the final stream to get the result.

Stream **does not store** the elements.

The aggregate operations that we perform on the collection, array or any other data source **do not change** the data of the source, they simply return a new stream.

All the stream operations are **lazy** in nature which means they are not executed until they are needed.

import java.util.ArrayList;

import java.util.List;

public class Example{

public static void main(String[] args) {

List<String> names = new ArrayList<String>();

names.add("Ajeet");

names.add("Negan");

names.add("Aditya");

names.add("Steve");

//Using Stream and Lambda expression

long count = names.stream().filter(str->str.length()<6).count();

System.out.println("There are "+count+" strings with length less than 6");

}

}

4.forEach() method

forEach is used method to iterate over collections and streams in java.

import java.util.List;

import java.util.ArrayList;

public class Example {

public static void main(String[] args) {

List<String> fruits = new ArrayList<String>();

fruits.add("Apple");

fruits.add("Orange");

fruits.add("Banana");

fruits.add("Pear");

fruits.add("Mango");

//lambda expression in forEach Method

fruits.forEach(str->System.out.println(str));

}

}

5.Java Time API

There was no standard approach or API in java for date and time in Java. One of the nice addition in Java 8 is the java.time package that will streamline the process of working with time in java.

It has some sub-packages java.time.format that provides classes to print and parse dates and times and java.time.zone provides support for time-zones and their rules.

The new Time API prefers enums over integer constants for months and days of the week. One of the useful class is DateTimeFormatter for converting datetime objects to strings.

Q2. Static and default methods of interface

.Default and static methods in interfaces

Java 8 allows the interfaces to have default and static methods. The reason we have default methods in interfaces is to allow the developers to add new methods to the interfaces without affecting the classes that implements these interfaces.

**Static methods** in interfaces are similar to the default methods except that we cannot override these methods in the classes that implements these interfaces.

interface MyInterface{

/\* This is a default method so we need not

\* to implement this method in the implementation

\* classes

\*/

default void newMethod(){

System.out.println("Newly added default method");

}

/\* This is a static method. Static method in interface is

\* similar to default method except that we cannot override

\* them in the implementation classes.

\* Similar to default methods, we need to implement these methods

\* in implementation classes so we can safely add them to the

\* existing interfaces.

\*/

static void anotherNewMethod(){

System.out.println("Newly added static method");

}

/\* Already existing public and abstract method

\* We must need to implement this method in

\* implementation classes.

\*/

void existingMethod(String str);

}

public class Example implements MyInterface{

// implementing abstract method

public void existingMethod(String str){

System.out.println("String is: "+str);

}

public static void main(String[] args) {

Example obj = new Example();

//calling the default method of interface

obj.newMethod();

//calling the static method of interface

MyInterface.anotherNewMethod();

//calling the abstract method of interface

obj.existingMethod("Java 8 is easy to learn");

}

}

Q3. forEach() Statement

Java provides a new method forEach() to iterate the elements. It is defined in Iterable and Stream interface. It is a default method defined in the Iterable interface. Collection classes which extends Iterable interface can use forEach loop to iterate elements.

This method takes a single parameter which is a functional interface. So, you can pass lambda expression as an argument.

**import** java.util.ArrayList;

**import** java.util.List;

**public** **class** ForEachExample {

**public** **static** **void** main(String[] args) {

        List<String> gamesList = **new** ArrayList<String>();

        gamesList.add("Football");

        gamesList.add("Cricket");

        gamesList.add("Chess");

        gamesList.add("Hocky");

        System.out.println("Iterating by passing lambda expression-");

        gamesList.forEach(games -> System.out.println(games));

    }

}

Q4. Exception Handling

An exception is an anomalous condition that alters or interrupts the flow of execution. Java provides built-in exception handling to deal with such conditions.

Exceptions and errors fall into three categories: checked exceptions, unchecked exceptions, and errors.

Checked Exceptions

* Checked exceptions are checked by the compiler at compile time.
* Methods that throw a checked exception must indicate so in the method declaration using the throws clause. This must continue all the way up the calling stack until the exception is handled.
* All checked exceptions must be explicitly caught with a catch block.
* Checked exceptions include exceptions of the type Exception, and all classes that are subtypes of Exception, except for RuntimeException and the subtypes of RuntimeException.

**Unchecked exceptions** − An unchecked exception is an exception that occurs at the time of execution. These are also called as **Runtime Exceptions**. These include programming bugs, such as logic errors or improper use of an API. Runtime exceptions are ignored at the time of compilation.

**Errors** − These are not exceptions at all, but problems that arise beyond the control of the user or the programmer. Errors are typically ignored in your code because you can rarely do anything about an error. For example, if a stack overflow occurs, an error will arise. They are also ignored at the time of compilation.