Java

JAVA 8 FEATURES:

-> Lambda expressions,

-> Method references,

-> Functional interfaces,

-> Stream API,

-> Default methods,

-> Base64 Encode Decode,

-> Static methods in interface,

-> Optional class,

-> Collectors class,

-> ForEach() method,

-> Nashorn JavaScript Engine,

-> Parallel Array Sorting,

-> Type and Repating Annotations,

-> IO Enhancements,

-> Concurrency Enhancements,

-> JDBC Enhancements etc.

Lambda Expression:

-> helps to write code in Functional style

-> implements Single Abstract Method (SAM) / Interface by using an expression

-> very useful in collection library in which it helps to iterate, filter and extract data

-> Java lambda expression is treated as a function, so compiler does not create .class file

-> Lambda expression is used to provide the implementation of an interface which has functional interface Functional Interface :

-> Lambda expression provides implementation of functional interface

-> An interface which has only one abstract method is called functional interface

-> Java provides an anotation @FunctionalInterface

Syntax : (argument-list) -> {body}

1) Argument-list: It can be empty or non-empty as well.

2) Arrow-token: It is used to link arguments-list and body of expression.

3) Body: It contains expressions and statements for lambda expression.

Example :

interface rectangle{  
 void draw();  
}  
  
  
public class LamdaExpression {  
 public static void main(String[] args) {  
 int l = 2;  
 rectangle d = ()-> {  
 System.*out*.println("Rectangle "+l);  
 };  
 d.draw();  
 }  
  
}

interface aj {  
 String ajay();  
}  
  
public class LamdaExpression {  
 public static void main(String[] args) {  
 aj s = ()-> {  
 return "hi";  
 };  
  
 System.*out*.println(s.ajay());  
 }  
}

public class LamdaExpression {  
 public static void main(String[] args) {  
 List<String> list = new ArrayList<String>();  
 list.add("ajay");  
 list.add("aj");  
 list.forEach(  
 (n)-> System.*out*.println(n)  
 );  
 }  
}

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

-> It is a new feature in Java, which helps to achieve functional programming approach

Annotations: @FunctionalInterface

// Example:

@java.lang.FunctionalInterface  
interface aj{  
 void ajay(String msg);  
}  
  
public class FunctionalInterface implements aj{  
 public void ajay(String msg){  
 System.*out*.println(msg);  
 }  
  
 public static void main(String[] args) {  
 FunctionalInterface f = new FunctionalInterface();  
 f.ajay("hi");  
 }  
  
}

Method Reference:

-> Java 8 Method reference is used to refer method of functional interface.

-> It is compact and easy form of lambda expression.

-> Each time when you are using lambda expression to just referring a method, you can replace your lambda expression with method reference.

// Example :

@java.lang.FunctionalInterface  
interface Display {  
 void display();  
}  
public class MethodReferences {  
 public void myMethod() {  
 System.*out*.println("method reference in java 8");  
 }  
 public static void main(String[] args) {  
 MethodReferences obj = new MethodReferences();  
 // Reference to the method using the object of the class myMethod  
 Display ref = obj::myMethod;  
 // Calling the method inside the functional interface Display  
 ref.display();  
 }  
}

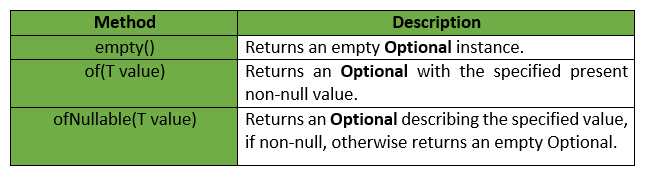
Optional :

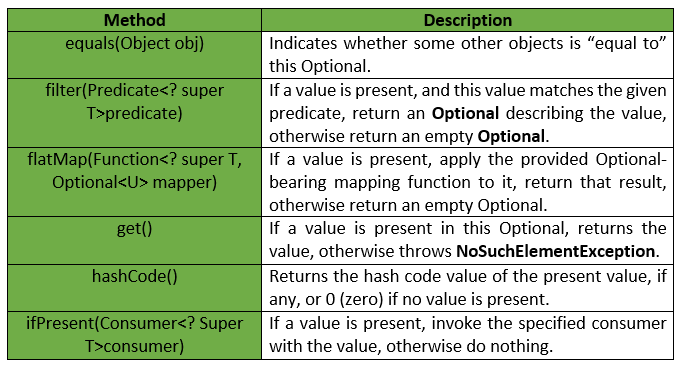
-> It is a public final class which is used to deal with NullPointerException in Java application.

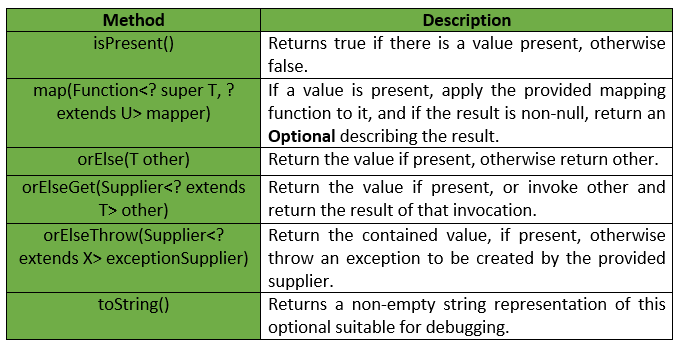
-> We must import java.util package to use this class.

-> It provides methods to check the presence of value for particular variable

The following table shows the list of Instance Methods provided by the Optional Class :







// Example :

public class OptionalDemo {  
 public static void main(String[] args)  
 {  
 String[] words = new String[10];  
 Optional<String> checkNull  
 = Optional.*ofNullable*(words[5]);  
 if (checkNull.isPresent()) {  
 String word = words[5].toLowerCase();  
 System.*out*.print(word);  
 }  
 else  
 System.*out*.println("word is null");  
 }  
}

Stream API :

-> Java 8 java.util.stream package consists of classes, interfaces and

-> an enum to allow functional-style operations on the elements.

-> It performs lazy computation. So, it executes only when it requires

public class StreamExample {  
 public static void main(String[] args){  
 Stream.*iterate*(1, element->element+1)  
 .filter(element->element%2==0)  
 .limit(10)  
 .forEach(System.*out*::println);  
 }  
}

//Example :

class Product{  
 int id;  
 String name;  
 float price;  
 public Product(int id, String name, float price) {  
 this.id = id;  
 this.name = name;  
 this.price = price;  
 }  
}  
public class StreamExample {  
 public static void main(String[] args) {  
 List<Product> productsList = new ArrayList<Product>();  
 //Adding Products  
 productsList.add(new Product(1,"HP Laptop",25000f));  
 productsList.add(new Product(2,"Dell Laptop",30000f));  
 productsList.add(new Product(3,"Lenevo Laptop",28000f));  
 productsList.add(new Product(4,"Sony Laptop",28000f));  
 productsList.add(new Product(5,"Apple Laptop",90000f));  
 List<Float> productPriceList2 =productsList.stream()  
 .filter(p -> p.price >25000 & p.price<30000)// filtering data  
 .map(p->p.price) // fetching price  
 .collect(Collectors.*toList*()); // collecting as list  
 System.*out*.println(productPriceList2);  
 }  
}

Collectors :

-> Collectors is a final class that extends Object class.

-> It provides reduction operations, such as accumulating elements into collections, summarizing elements according to various criteria, etc

Constructor :

-> constructor is a block of codes similar to the method.

-> It is called when an instance of the class is created.

-> At the time of calling constructor, memory for the object is allocated in the memory.

-> It is a special type of method which is used to initialize the object.

-> Every time an object is created using the new() keyword, at least one constructor is called.

-> It calls a default constructor if there is no constructor available in the class. In such case, Java compiler provides a default constructor by default.

Rules for creating Java constructor

-> Constructor name must be the same as its class name

-> A Constructor must have no explicit return type

-> A Java constructor cannot be abstract, static, final, and synchronized

Method :

-> A method is a block of code or collection of statements or

-> a set of code grouped together to perform a certain task or operation.

-> It is used to achieve the reusability of code. We write a method once and use it many times.

-> We do not require to write code again and again. It also provides the easy modification and readability of code, just by adding or removing a chunk of code.

-> The method is executed only when we call or invoke it.

-> The most important method in Java is the main() method.



There are two types of instance method:

1. Accessor Method - reads the instance variable (get /getter)

2. Mutator Method - read the instance variable(s) and also modify the values (set / setter)

Abstract Method

-> The method that does not has method body is known as abstract method.

-> In other words, without an implementation is known as abstract method.

-> It always declares in the abstract class.

-> It means the class itself must be abstract if it has abstract method.

-> To create an abstract method, we use the keyword abstract

Syntax : abstract void method\_name();

Four Pillars of Java :

1. Inheritance :

* child class can inherites all the properties of parent class.
* Multiple /hybrid inheritance is not possible in java
* Code reusability
* Can achieve polymorphism using inheritance

1. Polymorphism:

* Allows us to perform single action in multiple ways
* Compile time Polymorphism
  + Static polymorphism
  + Can achieve by method overloading
  + Compiler handles compile time polymorphism

Method Overloading :

* + Must have at least two methods with same name
  + Both methods should be in same class
  + Both methods must have different arguments
* Run time Polymorphism
  + Dynamic polymorphism
  + Achieved by method Overriding
  + JVM handles Run time polymorphism

Method Overriding :

* + Must have at least two methods with same name
  + Both methods should be in different class
  + Both methods must have same arguments

1. Abstraction :
   * + Hiding internal info and displaying necessary information
     + Achieved using abstractClassClass and interfaces
2. Encapsulation :

* Integrating all code into single unit
* Declare variables as private
* Provide getters and setters