Java 8 all important Features

Java 8 Features :

Oracle released a new version of Java 8 in March 18, 2014 . it was a revolutionary release of the java for software development platform . it includes various upgrades to the java programming , JVM , Tools and libraries.

Java 8 Programming Agenda:

01. Lambda Expressions
02. Functional interfaces
03. Method references
04. Stream API
05. Default methods
06. Base64 Encode Decode
07. Static method in interface
08. Optional class
09. Collectors class
10. ForEach() method
11. Nashorn Javascript Engine
12. Parallal array sorting
13. Type and Repating Annotations
14. IO Enhancements
15. Currency Enhancements
16. JDBC Enhancements.

Lambda Expression:

Lambda Expression is a new and important feature of Java . Which was Included in Java SE 8. It provides us a clear and concise way to represent one method interfacing using an expression. One method interfacing means that single abstract method (SAM). Which is a 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 annotation @FunctionalInterface , which is used to declare an interface as functional interface.

Why use Lambda Expression:

1. It provides implementation of Functional Interface.
2. It provides less coding

Java Lambda Expression Syntax	Describe
(argument-list)-> {body}	Argument-list : it can be empty or non-empty. Arrow-token : it used to link argument-list and body expression. Body : it contains expression and statements for lambda expression.

No Parameter Syntax	One Parameter Syntax	Two Parameter Syntax

()->{
no parameter lambda
}
	(p1)->{
	Single Parameter lambda
	}
		(p1,p2)->{
		Multiple parameter lambda
		}

What is lambda expression?

It is an anonymous function .

Anonymous means for :

1. Nameless
2. Without return type
3. Without modifiers

Example:

Example_1	With Lambda Expression	
		output

```java
package java_8_features_example_javapoint;

/*--firstly we declare a interface which name is Drawable1--*/
/*--and this example have a one single abstract method--*/

interface Drawable1 {
        public void draw();
}

public class Example_1 {
        public static void main(String[] args) {

                int width = 10;

/*--without lambda , Drawable1 implementation using anonymous class--*/

                Drawable1 d = new Drawable1() {

                        public void draw() {
                                System.out.println("Drawing : " + width);
                        }
                };

                d.draw();
        }
}
```

```
Drawing : 10
```

Example_2	With Lambda Expression

```java
package java_8_features_example_javapoint;

/*--firstly we declare a interface which name is Drawable2--*/
/*--and this example have a one single abstract method--*/

interface Drawable2 {
	public void draw();
}

public class Example_2 {
	public static void main(String[] args) {

		int width = 10;

		// here used in lambda expression
		Drawable2 d = () -> {
			System.out.println("Drawing : " + width);
			};

		d.draw();
		}
	}
}
```

output

```
Drawing : 10
```

Example_3	Lambda Expression No Parameter

```java
package java_8_features_example_javapoint;

interface SaySomething {
	public String say();
}

public class Example_3 {

	public static void main(String[] args) {
		SaySomething s = () -> {
			return "i have nothing to say";
		};

		System.out.println(s.say());
	}
}
```

output

```
i have nothing to say
```

Example_4 Lambda Expression Single Parameter

```java
package java_8_features_example_javapoint;

interface SaySomething1 {
        public String say(String name);
}

public class Example_4 {
        public static void main(String[] args) {
                SaySomething1 s = (name) -> {
                        return "Hello" + name + "Programming";
                };

                System.out.println(s.say(" java "));
        }
}
```

output

```
Hello java Programming
```

Example_5 Lambda Expression Multiple Parameter

```java
package java_8_features_example_javapoint;

interface Summition {
        public int Sum(int a, int b);
}

public class Example_5 {
        public static void main(String[] args) {

        /*--here , lambda expression use without data type--*/
        Summition s = (a, b) -> (a + b);
        System.out.println("result is = " + s.Sum(13, 12));

        /*--here , lambda expression use with data type--*/
        Summition s1 = (int a, int b) -> (a + b);
        System.out.println("result is = " + s1.Sum(15, 12));

        /*--here , lambda expression use changing variable--*/
        Summition s2 = (int f, int g) -> (f + g);
        System.out.println("result is = " + s.Sum(25, 12));
        }
}
```

output

```
result is = 25
result is = 27
result is = 37
```

Example_6 Lambda Expression Multiple Parameter

```java
package java_8_features_example_javapoint;

interface Addable {
        public int add(int a, int b);
}

public class Example_6 {
     public static void main(String[] args) {
                Addable ad = (int a, int b) -> {
                        return (a + b);
                };
        System.out.println("the result = " + ad.add(345, 505));
        }
}
```

output

```
the result = 850
```

Example_7 — Lambda Expression Example : For Each Loop

```java
package java_8_features_example_javapoint;

import java.util.*;

public class Example_7 {

        public static void main(String[] args) {
                List<String> list = new ArrayList();
                list.add("hello");
                list.add("java");
                list.add("programming");
                list.add("language");

                list.forEach(n -> System.out.println(n));
        }
}
```

output

```
hello
java
programming
language
```

Example_8 — Lambda Expression with Multiple Statement

```java
package java_8_features_example_javapoint;

@FunctionalInterface
interface SaySomething2 {
        String say(String message);
}

public class Example_8 {
        public static void main(String[] args) {
        // we can pass multiple statements in lambda expression
                SaySomething2 person = (message) -> {
                        String str1 = "i would like to say";
                        String str2 = str1 + message;
                        return str2;
                };

                System.out.println(person.say(",hi everyone"));
        }
}
```

output

If we declare @FunctionalInterface annotation before the interface class then interface class must be add single abstract method. Otherwise show compile time error.can not use two abstract method.

```
i would like to say,hi everyone
```

Example_9 — Lambda Expression Example : Creating Thread

```java
package java_8_features_example_javapoint;

public class Example_9 {

public static void main(String[] args) {
                /*--Thread Example without lambda--*/

        Runnable r1 = new Runnable() {
                public void run() {
                        System.out.println("Thread 1 is running");
                        }
                };
                Thread t1 = new Thread(r1);
                t1.start();
```

output

We can use lambda expression Creating thread.

```
Thread 1 is running
Thread 2 is running
```

```
		/*--Thread Example with lambda--*/
		Runnable r2 = () -> {
			System.out.println("Thread 2 is running");
		};
		Thread t2 = new Thread(r2);
		t2.start();
	}
}
```

Lambda Expression used in collection framework. It provides efficient and concise way to iterate , filter and fetch data.

Example_10	Lambda Expression Example : Collection comparator

```
package java_8_features_example_javapoint;

import java.util.*;

class Product {
	int id;
	String name;
	double price;

	public Product(int id, String name, double price) {
		super();
		this.id = id;
		this.name = name;
		this.price = price;
	}
}

public class Example_10 {
    public static void main(String[] args) {

	List<Product> list = new ArrayList<Product>();
	// add the product in list
	list.add(new Product(101, "Dell", 25000));
	list.add(new Product(105, "Asus", 65000));
	list.add(new Product(104, "Monitor", 6500));
	list.add(new Product(103, "keyboard", 350));
	list.add(new Product(102, "Ram", 1200));

	System.out.println("Sorting on the basis of name");

	//here, used lambda expression
	Collections.sort(list, (p1, p2) -> {
		return p1.name.compareTo(p2.name);
	   });

	for (Product p : list) {

	System.out.println(p.id + " " + p.name + " " + p.price);
		}
	}
}
```

output

```
Sorting on the basis of
name
105 Asus 65000.0
101 Dell 25000.0
104 Monitor 6500.0
102 Ram 1200.0
103 keyboard 350.0
```

Example_11	Lambda Expression Example Filter Collection data

```java
package java_8_features_example_javapoint;

import java.util.*;
import java.util.stream.*;

class Product1 {
	int id;
	String name;
	double price;

	public Product1(int id, String name, double price) {
		super();
		this.id = id;
		this.name = name;
		this.price = price;
	}
}

public class Example_11 {

	public static void main(String[] args) {
		List<Product1> list = new ArrayList<Product1>();
		// adding list value
		list.add(new Product1(101, "Dell", 25000));
		list.add(new Product1(105, "Asus", 65000));
		list.add(new Product1(104, "Monitor", 6500));
		list.add(new Product1(103, "keyboard", 350));
		list.add(new Product1(102, "Ram", 1200));

		// using lambda to filter data
Stream<Product1> filter_data = list.stream().filter(p -> p.price > 300
&& p.price < 25000);

filter_data.forEach(product -> System.out.println(product.id + " " +
product.name + " " + product.price));

	}
}
```

output

```
104 Monitor 6500.0
102 Ram 1200.0
103 keyboard 350.0
```

Java Functional Interfaces:

Java Functional interface that contains one abstract method. It can have any number of default method and static method. But can contain only one abstract method . it can also declare methods of object class.

Example_12	Lambda Expression Example : For Each Loop

```java
package java_8_features_example_javapoint;
interface FunctionalInterfaceRule {
	public void Say();//single abstract method
	default void show() {
		}// this is default method
	public static void display() {
		}// this is static method
}
```

output

Example_13	Lambda Expression Example : Functional Interface

```java
package java_8_features_example_javapoint;

@FunctionalInterface
interface SaySomething3 {
	void say(String message);
}

public class Example_13 implements SaySomething3 {
	public void say(String message) {
		System.out.println(message);
	}
	public static void main(String[] args) {
		Example_13 e = new Example_13();
		e.say("hello");
	}
}
```

output

```
hello
```

Java Predefined Functional Interface:

Example_14	Lambda Expression Example : Predicate

```java
package java_8_features_example_javapoint;
import java.util.function.*;

public class Example_14 {
	public static void main(String[] args) {
		Predicate<Integer> p = i -> i % 2 == 0;
		System.out.println(p.test(4));
		 System.out.println(p.test(5));
	}
}
```

output

It represents a predicate (Boolean-valued function) of one argument.

```
true
false
```

Example_15	Lambda Expression Example : Function

```java
package java_8_features_example_javapoint;
import java.util.function.*;

public class Example_15 {
	public static void main(String[] args) {

		Function<Integer, Integer> f=i->i*i;
		System.out.println("The Square = " +f.apply(4));
		System.out.println("The Square = " +f.apply(5));
	}
}
```

output

It represents a function that accept one argument and returns a result.

```
The Square = 16
The Square = 25
```

Example_16	Lambda Expression Example : Consume

```java
package java_8_features_example_javapoint;
import java.util.function.*;

public class Example_16 {
	public static void main(String[] args) {
		Consumer<String> con = x -> System.out.println(x);
		con.accept("java");
	}
}
```

output

It represents an operation that accepts a single argument and returns no result.

```
java
```

Example_17	Lambda Expression Example : Supplier

```
package java_8_features_example_javapoint;

import java.util.function.*;

class Student {
	private int id;
	private String name;
	private double age;

	public Student(int id, String name, double age) {
		super();
		this.id = id;
		this.name = name;
		this.age = age;
	}

	public int getId() {
		return id;
	}

	public void setId(int id) {
		this.id = id;
	}

	public String getName() {
		return name;
	}

	public void setName(String name) {
		this.name = name;
	}

	public double getAge() {
		return age;
	}

	public void setAge(double age) {
		this.age = age;
	}

	public String toString() {
		return "Student [id=" + id + ", name=" + name
+ ", age=" + age + "]";
	}
}

public class Example_17 {

	public static void main(String[] args) {
		Supplier stusup=()->new Student(101,"java",29);
		Student student=(Student)stusup.get();
		System.out.println(student);
	}
}
```

output

It represents a supplier of result.

```
Student [id=101,
name=java, age=29.0]
```

Example_18	Lambda Expression Example : BiConsumer

```java
package java_8_features_example_javapoint;
import java.util.function.*;
public class Example_18 {

public static void main(String[] args) {
     BiConsumer<Integer, Integer> bicon=(a,b)->System.out.println(a+b);
     bicon.accept(10, 5);
        }
}
```

output

It takes two arguments and returns nothing.

Java 8 Stream:

Java provides a new additional package in java 8 called java.util.stream. This Package consists of classes , interfaces and enum to allow functional style operation on the elements. We can use stream by importing java.util.stream package.

Stream does not store elements. It easily collect data from such as data structure , array or I/O Channel , and also that collected data through a pipeline of computational operations.

Stream is functional in nature. It can not modify the source . without removing collectors data, we can filter the elements.

It is lazy and evaluates code only when required.

We can use stream to filter , collect , print and convert one data structure to other .

Example_19	Stream Example : Filtering collections without stream

```java
package java_8_features_example_javapoint;
import java.util.*;

class Product3 {
        int id;
        String name;
        double price;
public Product3(int id, String name, double price) {
                super();
                this.id = id;
                this.name = name;
                this.price = price;
        }
}

public class Example_19 {
    public static void main(String[] args) {
        List<Product3> product_list = new ArrayList<Product3>();
                // adding list value
                product_list.add(new Product3(101, "Dell", 25000));
                product_list.add(new Product3(105, "Asus", 65000));
                product_list.add(new Product3(104, "Monitor", 6500));
                product_list.add(new Product3(103, "keyboard", 350));
                product_list.add(new Product3(102, "Ram", 1200));

        List<Double> product_price_list = new ArrayList<Double>();
```

output

```
[25000.0, 6500.0,
 350.0, 1200.0]
```

```java
                // filtering data from list
                for (Product3 product : product_list) {
                        if (product.price < 30000) {
                                // adding price to product_price_list
                                product_price_list.add(product.price);
                        }
                }

                System.out.println(product_price_list);

        }
}
```

Example_20	Stream Example : Filtering collections with stream

```java
package java_8_features_example_javapoint;
import java.util.*;
import java.util.stream.Collectors;

class Product4{
        int id;
        String name;
        double price;
        public Product4(int id, String name, double price) {
                super();
                this.id = id;
                this.name = name;
                this.price = price;
        }
}
public class Example_20 {

public static void main(String[] args) {
        List<Product4> product_list = new ArrayList<Product4>();
                // adding list value
                product_list.add(new Product4(101, "Dell", 25000));
                product_list.add(new Product4(105, "Asus", 65000));
                product_list.add(new Product4(104, "Monitor", 6500));
                product_list.add(new Product4(103, "keyboard", 350));
                product_list.add(new Product4(102, "Ram", 1200));

                List<Double> product_price_list=product_list.stream()
                        .filter(p->p.price<30000)//filtering data
                        .map(p->p.price)//fetching data
                        .collect(Collectors.toList());//collecting as list
                System.out.println(product_price_list);
        }
}
```

output

```
[25000.0, 6500.0,
 350.0, 1200.0]
```

Example_21	Stream Example : Filtering and Iterating Collection

```java
package java_8_features_example_javapoint;
import java.util.*;

class Product5 {
	int id;
	String name;
	double price;

	public Product5(int id, String name, double price) {
		super();
		this.id = id;
		this.name = name;
		this.price = price;
	}
}

public class Example_21 {
    public static void main(String[] args) {
	List<Product5> product_list = new ArrayList<Product5>();
		// adding list value
		product_list.add(new Product5(101, "Dell", 25000));
		product_list.add(new Product5(105, "Asus", 65000));
		product_list.add(new Product5(104, "Monitor", 6500));
		product_list.add(new Product5(103, "keyboard", 350));
		product_list.add(new Product5(102, "Ram", 1200));

		// This is more compact approach for filtering data
		product_list.stream()
		.filter(p->p.price==25000)
		.forEach(p->System.out.println(p.name));
	}
}
```

output

```
Dell
```