Unit-3

.Functional Interface

Functional Interface in Java enables users to implement functional programming in Java. In functional programming, the function is an independent entity.

Java is an object-oriented programming language i.e everything in java rotates around the java classes and their objects.

No function is independently present on its own in java. They are part of classes or interfaces. And to use them we require either the class or the object of the respective class to call that function.

Functional interfaces were introduced in Java 8. A functional interface can contain only one abstract method and it can contain any number of static and default (non-abstract) methods.

Functional Interface in Java is also called Single Abstract Method (SAM) interface. From Java 8 onwards, to represent the instance of functional interfaces, lambda expressions are used.

A functional interface is an interface that contains only one abstract method. It can have any number of default methods, static methods, and abstract methods from the Object class. Functional interfaces are used primarily to enable functional programming techniques in Java, particularly with lambda expressions and method references.

```java
@FunctionalInterface

interface Square {
  int calculate(int x);
}

class Test {
  public static void main(String args[])
  {
    int a = 5;

    // lambda expression to define the calculate method
    Square s = (int x) -> x * x;

    // parameter passed and return type must be
    // same as defined in the prototype
    int ans = s.calculate(a);
    System.out.println(ans);
  }
}
```

Some Built-in Functional interface in Java

There are many interfaces that are converted into functional interfaces. All these interfaces are annotated with @FunctionalInterface. These interfaces are as follows –

- **Runnable** –> This interface only contains the run() method.
- **Comparable** –> This interface only contains the compareTo() method.
- **ActionListener** –> This interface only contains the actionPerformed() method.
- **Callable** –> This interface only contains the call() method.

In Java four main kinds of functional interfaces which can be applied in multiple situations as mentioned below:

Consumer, Predicate, Function, Supplier

Advantages of Functional Interface in Java:

- **Flexibility:** Functional interfaces offer flexibility by allowing methods to be passed as parameters, facilitating the implementation of various behaviors without the need for multiple interfaces.
- **Conciseness:** They promote concise code by enabling the use of lambda expressions, reducing verbosity and enhancing readability.

- **Parallelism:** Functional interfaces can facilitate parallelism and concurrency, as they often represent operations that can be easily parallelized, such as mapping or reducing functions.
- **Composition:** They support function composition, enabling the creation of complex behaviors by combining simpler functions, leading to more modular and maintainable code.
- **Functional Programming Paradigm:** They align with the principles of functional programming, promoting immutability, referential transparency, and other beneficial characteristics.

Disadvantages of Functional Interface in Java:

- **Learning Curve:** For developers unfamiliar with functional programming concepts, understanding and utilizing functional interfaces effectively can pose a steep learning curve.
- **Complexity:** In some cases, the use of functional interfaces and lambda expressions can introduce complexity, especially in scenarios where there is a mix of imperative and functional programming styles within the same codebase.
- **Performance Overhead:** While modern JVMs optimize the performance of lambda expressions and functional interfaces, there may still be a slight performance overhead compared to traditional imperative code in certain scenarios.
- **Debugging Challenges:** Debugging code involving complex functional interfaces and lambda expressions can be challenging, as it may require understanding the behavior of higher-order functions and closures.
- **Compatibility:** Maintaining compatibility with older versions of Java or other programming languages that do not support functional interfaces may be a concern, particularly when integrating with legacy systems or libraries.

Java Lambda Expressions

The lambda expression was introduced first time in Java 8. Its main objective to increase the expressive power of the language.

Lambda expression is, essentially, an anonymous or unnamed method. The lambda expression does not execute on its own. Instead, it is used to implement a method defined by a functional interface.

How to define lambda expression in Java?

```
(parameter list) -> lambda body
```

The new operator (->) used is known as an arrow operator or a lambda operator. Suppose, we have a method like this:

```
double getPiValue() {
    return 3.1415;
}
```

We can write this method using lambda expression as:

```
() -> 3.1415
```

Here, the method does not have any parameters.

Java SE 8 came with three big new features

1. Lambda Expressions
2. Stream API
3. Date API

In the same way, Java SE 9 is coming with three big features

1. Java Module System (Jigsaw Project)
2. Java REPL
3. Milling Project Coin

Java Module System provides an additional layer of encapsulation to our programs as we can specify which package can be utilized by the modules, and which modules could have entry to to them.

Difference Between Module and Package

The principal difference between the module and package is given below.

1. ***Module***: In easy words, we can say that the module is a set of related applications or It is a collection of related applications such that it affords an API handy to different modules that are internal and encapsulated.

Suppose permits take an example from the built-in module in Java let's take java.util for example. If we expect java.util as a module we recognize that there are a variety of instructions and sub-packages inside the java.util. Now we've assumed that java.util is a package deal the modules could be like java.util.Collections and java.util.stream.

2. Package: A package is a set of classes, interfaces, and sub-packages that are similar. There are mainly two types of packages in Java, they are:

- ***User Defined packages***: The packages that contain the classes or interfaces which are built based on the user and it is nothing but they are just defined by the user.
- ***Built-In Packages***: The packages that come pre-installed when we configure the Java in our system are called the built-in packages. For example, as we specified earlier such as java.net, java.awt, javax.swing, java.sql etc

Java 9 has one of the major changes in its features which is the ***Java module System.*** The main aim of the system is to collect Java packages and code to be collected into a single unit called a Module.

Advantages of Java SE 9 Module System

Java SE 9 Module System is going to provide the following benefits

- As Java SE 9 is going to divide JDK, JRE, JARs etc, into smaller modules, we can use whatever modules we want. So it is very easy to scale down the Java Application to Small devices.
- Ease of Testing and Maintainability.
- Supports better Performance.
- As public is not just public, it supports very Strong Encapsulation. We cannot access Internal Non-Critical APIs anymore.
- Modules can hide unwanted and internal details very safely, we can get better Security.
- Application is too small because we can use only what ever modules we want.
- Its easy to support Less Coupling between components.
- Its easy to support Single Responsibility Principle (SRP).

Diamond syntax

The Diamond operator makes code readability easier in the world of Java generics. Generics, introduced in Java 5, Permit the development of methods and classes that operate on any data.

The Diamond operator, denoted by `<>,` enhances code conciseness by inferring generic types during object instantiation.

In Java generics, T, E, and K are conventions for type parameters, acting as placeholders for actual types that will be specified later.

They enhance code reusability and type safety.

- T typically represents a generic type. It's commonly used when the specific type is not predetermined and can be any class or interface.
- E usually stands for element, often seen in collections like List<E> or Set<E>, indicating the type of elements the collection holds.
- K represents a key, primarily used in Map<K, V> to denote the type of keys in the map.

For instance,

1. Type Parameters:

- Placeholders for actual types are represented by letters like T, E, or K.
- It is specified within angle brackets (<>).
- *Example:* ArrayList<String> defines a list that can only hold strings.

2. Generic Classes and Interfaces:

- Classes and interfaces that can work with different data types through type parameters.

Examples:

`ArrayList<T>, HashMap<K, V>, Comparable<T>.`

3. Generic Methods:

- Techniques that apply to various kinds of data based on their type parameters.
- *Example:* public static <T> void swap(T[] arr, int i, int j) can swap elements in any array.

4. Bounds:

- There are limitations on the acceptable types when using the type parameter.
- Ensure that types have certain functionalities or relationships.
- *Examples:* extends Number, super Comparable<T>.

5. Diamond Operator (<>):

Syntactic shortcut for specifying type arguments when they can be inferred from the context.

Example:

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

6. Benefits of Generics:

- *Type Safety:* Eliminates the need for casting and reduces runtime errors.
- *Code Reusability:* Generic classes and methods can be reused differently to promote code efficiency.
- *Readability:* Improves code clarity by clearly indicating the intended types.
- *Type Inference:* The compiler can often infer type arguments, reducing code verbosity.

Generics are a powerful tool in Java that significantly enhances code quality, safety, and maintainability.

7. Generic Class

When creating generic classes, we specify the parameter types using <>, just like in C++. Use the following syntax to create generic class objects.

```
// Create a generic class instance
BaseType <Type> obj = new BaseType <Type>()
```

Diamond Operator Limitations:

The Diamond operator (<>) introduced in Java 7 has limitations. It can't be used with anonymous inner classes.

```
Map<String, List<Integer>> map = new HashMap<>() { /* Error in Java 7 */ };
```

Diamond operator for anonymous inner-class

Previously unusable for anonymous classes, the diamond operator is now improved:

Example:

```
MyClass<> obj = new MyClass<>() {}; // Works in Java 9
```

```java
// Program to illustrate the problem
// while linking diamond operator
// with an anonymous inner class

abstract class Geeksforgeeks<T> {
    abstract T add(T num1, T num2);
}

public class Geeks {
    public static void main(String[] args)
    {
        Geeksforgeeks<Integer> obj = new Geeksforgeeks<>() {
            Integer add(Integer n1, Integer n2)
            {
                return (n1 + n2);
            }
        };
        Integer result = obj.add(10, 20);
        System.out.println("Addition of two numbers: " + result);
    }
}
```

Output:

```
prog.java:9: error: cannot infer type arguments for Geeksforgeeks
      Geeksforgeeks  obj = new Geeksforgeeks  () {
                                                ^
  reason: cannot use '' with anonymous inner classes
  where T is a type-variable:
    T extends Object declared in class Geeksforgeeks
1 error
```

Java developer extended the feature of the diamond operator in JDK 9 by allowing the diamond operator to be used with anonymous inner classes too. If we run the above code with JDK 9, then code will run fine and we will generate the below output.

Output:

```
Addition of two numbers: 30
```

Local Variable Type Inference

Local Variable Type Inference is one of the most evident change to language available from Java 10 onwards.

It allows to define a variable using var and without specifying the type of it. The compiler infers the type of the variable using the value provided.

This type inference is restricted to local variables

Old way of declaring local variable.

String name = "Welcome to tutorialspoint.com";

New Way of declaring local variable.

var name = "Welcome to tutorialspoint.com";

Now compiler infers the type of name variable as String by inspecting the value provided.

Ex:

```
import java.util.List;

public class Tester {
   public static void main(String[] args) {
      var names = List.of("Julie", "Robert", "Chris", "Joseph");
      for (var name : names) {
         System.out.println(name);
      }
      System.out.println("");
      for (var i = 0; i < names.size(); i++) {
         System.out.println(names.get(i));
      }
   }
}
```

Output

It will print the following output.

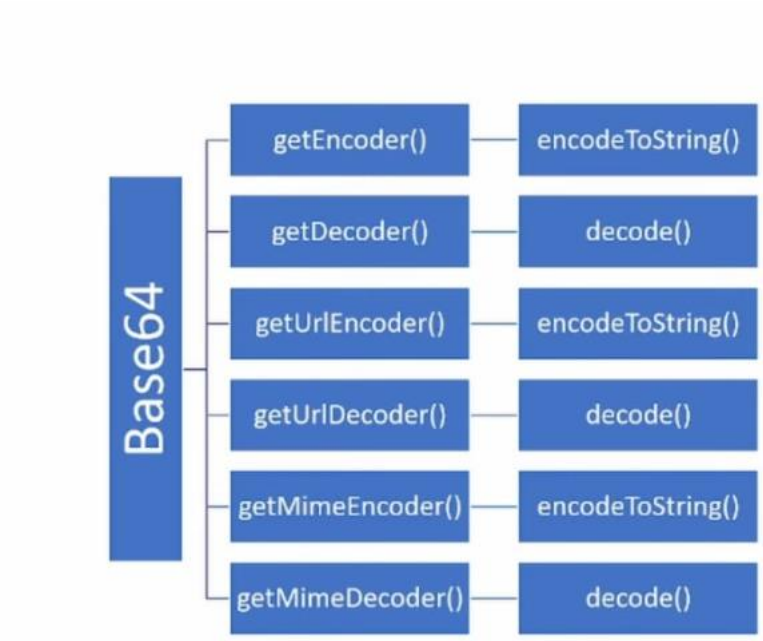
```
Julie
Robert
Chris
Joseph

Julie
Robert
Chris
Joseph
```

Base 64

Base 64 is an encoding scheme that converts binary data into text format so that encoded textual data can be easily transported over network un-corrupted and without any data loss.

The basic encoding means no line feeds are added to the output and the output is mapped to a set of characteristics in A-Za-z0-9+/



Example 1:

```
import java.util.Base64;

class Main
{
        public static void main(String[] args)
        {
                String msg=" This is a ex";
```

```
        String
    encmsg=Base64.getEncoder().encodeToString(msg.ge
    tBytes());
        System.out.println("Encrypted message
    is :"+encmsg);
    }
}
```

Example 2 :

```
import java.util.Base64;

class Main
{
    public static void main(String[] args)
    {
        String msg="This is an ex";
        String
    encmsg=Base64.getEncoder().encodeToString(msg.ge
    tBytes());
        System.out.println("Encrypted message
    is :"+encmsg);
        byte
    []dm=Base64.getDecoder().decode(encmsg);
        String decmsg=new String(dm);
        System.out.println("Decrypted message
    is :"+decmsg);

    }
}
```

Example 3:

```
import java.util.Base64;

class Main
{
    public static void main(String[] args)
    {
        String url=" http://www.google.com";
```

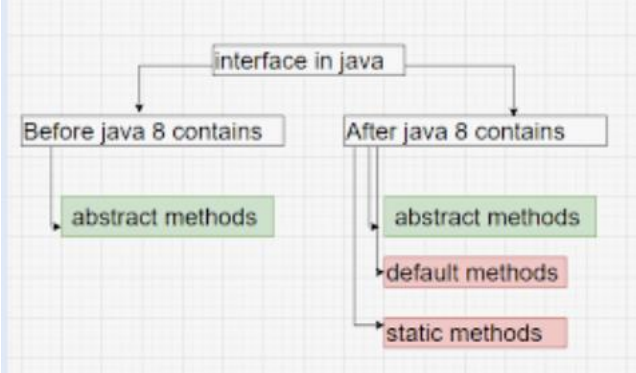
```
        String
    encurl=Base64.getUrlEncoder().encodeToString(url.ge
    tBytes());
        System.out.println("Encrypted Url is :"+encurl);
    }
}
```

Example 4:

```
import java.util.Base64;
class Main
{
    public static void main(String[] args)
    {
        String url=" http://www.google.com";
        String
    encurl=Base64.getUrlEncoder().encodeToString(url.ge
    tBytes());
        System.out.println("Encrypted Url is :"+encurl);

        byte[]
    du=Base64.getUrlDecoder().decode(encurl);
        String decurl=new String(du);
        System.out.println("Decrypted Url is :"+decurl);
    }
}
```

Difference Between Default and Static Interface Method in Java 8



Sr. No.	Key	Static Interface Method	Default Method
1	Basic	It is a static method which belongs to the interface only. We can write implementation of this method in interface itself	It is a method with default keyword and class can override this method
2	Method Invocation	Static method can invoke only on interface class not on class.	It can be invoked on interface as well as class
3	Method Name	Interface and implementing class , both can have static method with the same name without overriding each other.	We can override the default method in implementing class
4.	Use Case	It can be used as a utility method	It can be used to provide common functionality in all implementing classes

JAVA SWITCH EXPRESSIONS

Using **switch expressions**, we can return a value and we can use them within statements like other expressions. We can use **case L -> label** to return a value or using yield, we can return a value from a switch expression.

Java 12 introduces **expressions to Switch statements** and releases them as a preview feature. Java 13 added a new yield construct to return a value from a switch statement. With Java 14, **switch expression** now is a standard feature.

Switch Expression Using "case L ->" Labels

Java provides a new way to return a value from a switch expression using case L - > notation.

Syntax

```
case label1, label2, ..., labeln -> expression;|throw-
statement;|block
```

Ex 1 :

```
class Main
{
	public static void main(String[] args)
	{
		int a,b,c;
		a=10;b=2;c=1;

		switch(c)
		{
			case 1:
				System.out.println(a+b);
				break;
			case 2:
				System.out.println(a-b);
				break;
			default:
				System.out.println("invali");

				}
		}
}
```

Java 8

Ex 2:

```
class Main
{
	public static void main(String[] args)
	{
		int a,b;
		a=10;b=2;
		String c=”add”;

		switch(c)
		{
			case “add”:
				System.out.println(a+b);
				break;
			case “sub”:
				System.out.println(a-b);
				break;
			default:
				System.out.println("invali");

				}
		}
}
```

Before Switch Expression

Ex 3:

```
class Main
{
	public static void main(String[] args)
	{
		int a,b;
		a=10;b=2;
		enum c{add,sub}
		switch(c.add)
		{
			case add:
				System.out.println(a+b);
				break;
			case sub:
				System.out.println(a-b);
				break;
			default:
				System.out.println("invalid");
```

```
                                        }
                        }
}
```

Ex 4:

Switch Expression

```
class Main
{
        public static void main(String[] args)
        {
                int a,b,r;
                a=10;b=2;
                enum c{add,sub};
                r=switch(c.sub){

                        case add->a+b;
                        case sub->a-b;
                        default->-1;
                };
                                System.out.println("Result ="+r);


                }
}
```

Ex :5:

```
class Main
{
        public static void main(String[] args)
        {
                String day="wed";
                String r=switch(day)
                {
                        case "mon","tue","wed","thu","fri"->"working day";
                        case "sat","sun"->"non-working day";
                        default->"invalid day";
                };

                System.out.println("Result ="+r);


                }
}
```

Ex 6 : yield keyword

```
class Main
{
	public static void main(String[] args)
	{
		int a=10, b=2, c;
		String ch, r;
		ch="sub";
		r=switch(ch)
		{
			case "add"->{
				c=a+b;
				yield "sum="+c;
				}
			case "sub"->{
				c=a-b;
				yield "sub="+c;
				}
			default->{
				yield "invali";
				}
		};
		System.out.println(r);
	}
}
```

Static method and Default Method

Default Method: Methods in an interface that have a default implementation.
They provide a default implementation that can be overridden if needed.

Syntax:

```
interface MyInterface
{
	default void myDefaultMethod()
	{
		System.out.println(“Default method “);
	}
}
```

Static method: A **static method** is a method that is associated with the interface in which it is defined rather than with any object.
Static method in interface are part of the interface class can't implement or override it whereas class can override the default method.

Syntax:

```
interface MyInterface
{
        static void myStaticMethod()
        {
                System.out.println("Static method ");
        }
}
```

Sr. No.	Key	Static Interface Method	Default Method
1	Basic	It is a static method which belongs to the interface only. We can write implementation of this method in interface itself	It is a method with default keyword and class can override this method
2	Method Invocation	Static method can invoke only on interface class not on class.	It can be invoked on interface as well as class
3	Method Name	Interface and implementing class , both can have static method with the same name without overriding each other.	We can override the default method in implementing class
4.	Use Case	It can be used as a utility method	It can be used to provide common functionality in all implementing classes

Ex1 : Default Method in Java

```
interface MyInterface
{

   public void cube(int x); // abstract method

   default void print() // default method
   {
     System.out.println("Hello I am default method!");
   }
}

class MyClass implements MyInterface
{
   // implementation of cube abstract method
   public void cube(int x)
   {
      System.out.println(x*x*x);
   }

   public static void main(String args[])
   {
      MyClass obj = new MyClass();
      obj.cube(4);

      obj.print(); // default method executed
   }
}
```

Output:

```
64
Hello I am default method!
```

Ex2 : Static method

```
interface MyInterface
{
      public void cube(int x); // abstract method

   static  void print() // static method
   {
     System.out.println("Hello I am static method!");
   }
}
```

```
class MyClass implements MyInterface
{
    // implementation of cube abstract method
    public void cube(int x)
    {
        System.out.println(x*x*x);
    }

    public static void main(String args[])
    {
        MyClass obj = new MyClass();
        obj.cube(4);

        MyInterface.print(); // static method executed
    }
}
```

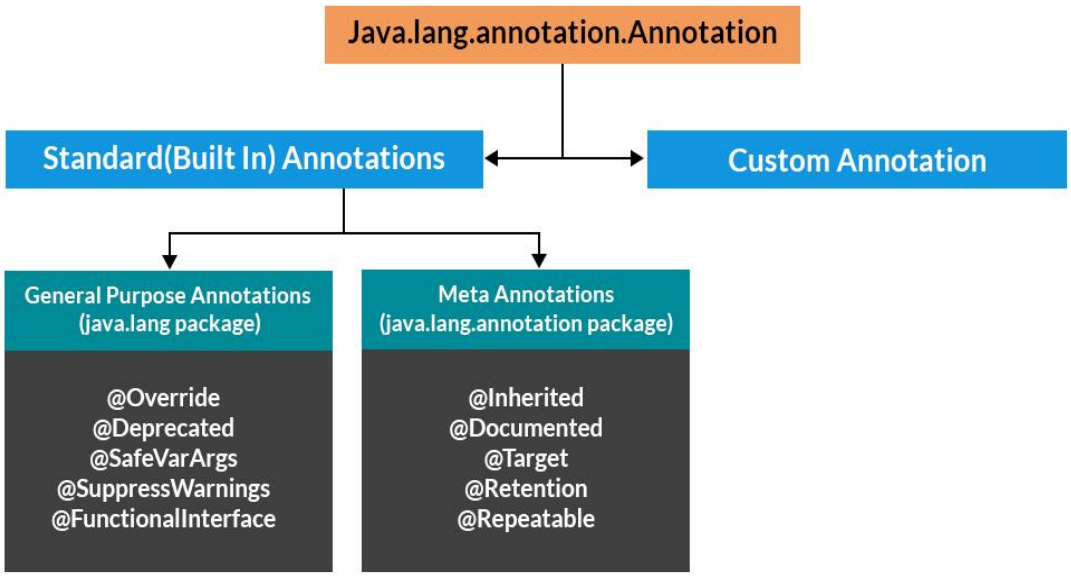
Output:
64
Hello I am static method!

Type Annotations

Annotations are used to provide supplemental information about a program.

- Annotations start with '**@**'.
- Annotations do not change the action of a compiled program.
- Annotations help to associate *metadata* (information) to the program elements i.e. instance variables, constructors, methods, classes, etc.
- Annotations are not pure comments as they can change the way a program is treated by the compiler. See below code for example.
- Annotations basically are used to provide additional information, so could be an alternative to XML and Java marker interfaces.
-

Hierarchy of Annotations in Java



In Java 8, two significant enhancements were introduced regarding annotations: type annotations and repeating annotations.

Type Annotations

Before Java 8, annotations could only be applied to declarations. Java 8 expanded their scope, allowing them to be applied to type uses. This means annotations can now appear anywhere a type is used, such as in:

- Class instance creation expressions (new)
- Casts
- implements clauses
- throws clauses
- Generic type arguments
- Array component types

Type annotations facilitate improved type checking and allow for the creation of pluggable type systems. For example, a custom plug-in could be written to ensure that a particular variable is never assigned null, preventing NullPointerException errors.

Java

```
@NonNull String str;
```

In this example, @NonNull is a type annotation indicating that the str variable should never be null.

Repeating Annotations

Prior to Java 8, it was not possible to apply the same annotation multiple times to a declaration. Java 8 introduced repeating annotations to address this limitation. To define a repeating annotation, the @Repeatable annotation is used. A container annotation is also required to hold the repeating annotations.

Java

```
@interface Color {
   String name();
}

@Repeatable(Colors.class)
@interface Color {
   String name();
}

@interface Colors {
   Color[] value();
}

@Color(name = "red")
@Color(name = "blue")
@Color(name = "green")
class Shirt {
}
```

In this example, @Color is a repeating annotation, and @Colors acts as its container. The Shirt class is annotated with multiple @Color annotations.

The Java compiler handles the wrapping of repeating annotations into their container annotation, providing a cleaner syntax for developers.

// Java Program to Demonstrate that Annotations are Not Barely Comments

```
// Class 1
class Base {
```

```
    // Method
    public void display()
    {
        System.out.println("Base display()");
    }
}

// Class 2
// Main class
class Derived extends Base {

    // Overriding method as already up in above class
    @Override public void display(int x)
    {
        // Print statement when this method is called
        System.out.println("Derived display(int )");
    }

    // Method 2
    // Main driver method
    public static void main(String args[])
    {
        // Creating object of this class inside main()
        Derived obj = new Derived();

        // Calling display() method inside main()
        obj.display();
    }
}
```

Output:

```
10: error: method does not override or implement
    a method from a supertype
```

If we remove parameter (int x) or we remove @override, the program compiles fine.

EX 2:

```java
// Java Program to Demonstrate Type Annotation

// Importing required classes
import java.lang.annotation.ElementType;
import java.lang.annotation.Target;

// Using target annotation to annotate a type
@Target(ElementType.TYPE_USE)

// Declaring a simple type annotation
@interface TypeAnnoDemo{}

// Main class
public class GFG {

    // Main driver method
    public static void main(String[] args) {

        // Annotating the type of a string
        @TypeAnnoDemo String string = "I am annotated with a type annotation";
        System.out.println(string);
        abc();
    }

    // Annotating return type of a function
    static @TypeAnnoDemo int abc() {

        System.out.println("This function's  return type is annotated");

        return 0;
    }
}
```

Output:

```
I am annotated with a type annotation
This function's  return type is annotated
```

Ex 2 :

```
// Java Program to Demonstrate a Repeatable Annotation

// Importing required classes
import java.lang.annotation.Annotation;
import java.lang.annotation.Repeatable;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.reflect.Method;

// Make Words annotation repeatable
@Retention(RetentionPolicy.RUNTIME)
@Repeatable(MyRepeatedAnnos.class)
@interface Words
{
   String word() default "Hello";
   int value() default 0;
}

// Create container annotation
@Retention(RetentionPolicy.RUNTIME)
@interface MyRepeatedAnnos
{
   Words[] value();
}
public class Main {

   // Repeat Words on newMethod
   @Words(word = "First", value = 1)
   @Words(word = "Second", value = 2)
   public static void newMethod()
   {
      Main obj = new Main();

      try {
         Class<?> c = obj.getClass();

         // Obtain the annotation for newMethod
         Method m = c.getMethod("newMethod");
```

```
            // Display the repeated annotation
            Annotation anno
               = m.getAnnotation(MyRepeatedAnnos.class);
            System.out.println(anno);
         }
         catch (NoSuchMethodException e) {
            System.out.println(e);
         }
      }
      public static void main(String[] args) { newMethod(); }
   }
```

Output:

```
@MyRepeatedAnnos(value={@Words(value=1, word="First"), @Words(value=2, word="Second")})
```

Text Block in Java

A text block is a multi-line string literal and the feature offers a clean way to format the string in a predictable way, without using most of the escape sequences. It starts and ends with a """ (three double-quotes marks) e.g.

```
public class Main {
   public static void main(String[] args)
   {
      String text = """
            <html>
               <head>
                  <title>Hello World</title>
               </head>
               <body>
                  Java rocks!
               </body>
            </html>""";

      System.out.println(text);
   }
}
```

Output:

```
<html>
   <head>
```

```
        <title>Hello World</title>
    </head>
    <body>
        Java rocks!
    </body>
</html>
```

With the traditional string representation, the code would look like

```
public class Main {
    public static void main(String[] args) {
        String text = "<html>\n"
                + " <head>\n"
                + "     <title>Hello World</title>\n"
                + " </head>\n"
                + " <body>\n"
                + "     Java rocks!\n"
                + " </body>\n"
                + "</html>";

        System.out.println(text);
    }
}
```

Another key difference is that a text block begins with *three double-quote characters followed by a line terminator* which is not the case with the traditional string representation.

It means

1. The text block can not be put on a single line.
2. The content of the text block can not follow the three opening double quotes on the same line.

```
String str = "Hello World!"; // The traditional string
String textBlock = """
       Hello World!
       """; // The text block
String notAllowed = """Hello World!"""; // Error
// The following code will fail compilationString notAllowed2
="""Hello
       World!
       """;
```

What Are Sealed Classes?

Sealed classes was introduced in Java 15. Sealed classes allow you to restrict which classes can extend or implement them. This provides a way to control the class hierarchy, ensuring that only specific, predefined classes can be part of that hierarchy. Or It is a techniques that limits the number of classes that can inherit the given class.

In below example, Shape is a sealed class, meaning only Circle, Rectangle, and Square can extend it. This ensures the hierarchy is controlled and prevents unintended classes from becoming part of it.

```
sealed class Shape permits Circle, Rectangle
{
        public void area()
        {
                System.out.println("I am Shape");
        }
}
non-sealed class Circle extends Shape
{
    public void  area()
        {
                System.out.println("I am Circle");
        }
```

```
}
 non-sealed class Rectangle extends Shape
{
     public void  area()
        {
                System.out.println("I am Rectangle");
        }
 }
class Main
{
        public static void main(String[] args)
        {
                Shape s1=new Circle();
                Shape s2=new Rectangle();
                s1.area();
                s2.area();
        }
}
```

What Are Records?

Records were introduced in Java 17. Records provide a compact syntax for declaring classes that are intended to be simple data carriers. They automatically generate methods like toString(), equals(), and hashCode().

With records, you can create immutable data carriers with minimal boilerplate code. The Person record in below example automatically provides methods to access its fields and a proper toString() implementation.

```
public record Person(String name, int age) {}
```
=> save above file as Person.java

```
public class Main {
    public static void main(String[] args) {
        Person p = new Person("John", 30);

        System.out.println("Name: " + p.name());
        System.out.println("Age: " + p.age());
        System.out.println(p);
    }
}
```