UNIT 3

NEW FEATURES

OF

JAVA

Java Functional Interfaces

An Interface that contains exactly one abstract method is known as functional interface. It can have any number of default, static methods but can contain only one abstract method. It can also declare methods of object class.

Functional Interface is also known as Single Abstract Method Interfaces or SAM Interfaces. It is a new feature in Java, which helps to achieve functional programming approach.

Functional Interface is additionally recognized as **Single Abstract Method Interfaces**. In short, they are also known as **SAM interfaces**. Functional interfaces in Java are the new feature that provides users with the approach of fundamental programming.
Functional interfaces are included in Java SE 8 with Lambda expressions and Method references in order to make code more readable, clean, and straightforward. Functional interfaces are interfaces that ensure that they include precisely only one abstract method. Functional interfaces are used and executed by representing the interface with an **annotation called *@FunctionalInterface***. As described earlier, functional interfaces can contain only one abstract method. However, they can include any quantity of default and static methods.

Java Functional Interfaces

A **functional interface** is an interface that contains only one abstract method. They can have only one functionality to exhibit. From Java 8 onwards, lambda expressions can be used to represent the instance of a functional interface. A functional interface can have any number of default methods. ***Runnable***, ***ActionListener***, *and* ***Comparable*** are some of the examples of functional interfaces.
Functional Interface is additionally recognized as **Single Abstract Method Interfaces**. In short, they are also known as **SAM interfaces**. Functional interfaces in Java are the new feature that provides users with the approach of fundamental programming.
Functional interfaces are included in Java SE 8 with Lambda expressions and Method references in order to make code more readable, clean, and straightforward. Functional interfaces are interfaces that ensure that they include precisely only one abstract method. Functional interfaces are used and executed by representing the interface with an **annotation called *@FunctionalInterface***. As described earlier, functional interfaces can contain only one abstract method. However, they can include any quantity of default and static methods.
In Functional interfaces, there is no need to use the abstract keyword as it is optional to use the abstract keyword because, by default, the method defined inside the interface is abstract only. We can also call Lambda expressions as the instance of functional interface.

Some Built-in Java Functional Interfaces

Since Java SE 1.8 onwards, 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.

Java SE 8 included four main kinds of functional interfaces which can be applied in multiple situations as mentioned below:

1. **Consumer**
2. **Predicate**
3. **Function**
4. **Supplier**

Example:-

```java
@FunctionalInterface
interface sayable{
    void say(String msg);
}
public class FunctionalInterfaceExample implements sayable{
    public void say(String msg){
        System.out.println(msg);
    }
    public static void main(String[] args) {
        FunctionalInterfaceExample fie = new FunctionalInterfaceExample();
        fie.say("Hello there");
    }
}
```

Java Lambda Expressions

Lambda expression is a new and important feature of Java which was included in Java SE 8. It provides a clear and concise way to represent one method interface using an expression. It is very useful in collection library. It helps to iterate, filter and extract data from collection.

The Lambda expression is used to provide the implementation of an interface which has functional interface. It saves a lot of code. In case of lambda expression, we don't need to define the method again for providing the implementation. Here, we just write the implementation code.

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

Lambda expression provides implementation of *functional interface*. An interface which has only one abstract method is called functional interface. Java provides an anotation *@FunctionalInterface*, which is used to declare an interface as functional interface.

Java lambda expression is consisted of three components.

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.

Java Lambda Expression Example: No Parameter

```java
interface Sayable{
    public String say();
}
public class LambdaExpressionExample3{
public static void main(String[] args) {
    Sayable s=()->{
        return "I have nothing to say.";
    };
    System.out.println(s.say());
}
}
```

Java Lambda Expression Example: Single Parameter

```java
interface Sayable{
    public String say(String name);
}

public class LambdaExpressionExample4{
    public static void main(String[] args) {

        // Lambda expression with single parameter.
        Sayable s1=(name)->{
            return "Hello, "+name;
        };
        System.out.println(s1.say("Sonoo"));

        // You can omit function parentheses
        Sayable s2= name ->{
            return "Hello, "+name;
        };
        System.out.println(s2.say("Sonoo"));
    }
}
```

Java Lambda Expression Example: Multiple Parameters

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

public class LambdaExpressionExample5{
    public static void main(String[] args) {

        // Multiple parameters in lambda expression
        Addable ad1=(a,b)->(a+b);
        System.out.println(ad1.add(10,20));

        // Multiple parameters with data type in lambda expression
        Addable ad2=(int a,int b)->(a+b);
        System.out.println(ad2.add(100,200));
    }
}
```

Java Method References

Java provides a new feature called method reference in 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.

Types of Method References

There are following types of method references in java:

1. Reference to a static method.
2. Reference to an instance method.
3. Reference to a constructor.

1) Reference to a Static Method

You can refer to static method defined in the class. Following is the syntax and example which describe the process of referring static method in Java.

Syntax

```
ContainingClass::staticMethodName
```

In the following example, we have defined a functional interface and referring a static method to it's functional method say().

```java
interface Sayable{
    void say();
}
public class MethodReference {
    public static void saySomething(){
        System.out.println("Hello, this is static method.");
    }
    public static void main(String[] args) {
        // Referring static method
                Sayable sayable = MethodReference::saySomething;
                // Calling interface method
                sayable.say();
            }
}
```

2) Reference to an Instance Method

like static methods, you can refer instance methods also. In the following example, we are describing the process of referring the instance method.

Syntax

```
containingObject::instanceMethodName
```

Example

In the following example, we are referring non-static methods. You can refer methods by class object and anonymous object.

```java
interface Sayable{
    void say();
}
public class InstanceMethodReference {
    public void saySomething(){
        System.out.println("Hello, this is non-static method.");
    }
    public static void main(String[] args) {
        InstanceMethodReference methodReference = new InstanceMethodReference(); // Creating object
            // Referring non-static method using reference
                Sayable sayable = methodReference::saySomething;
            // Calling interface method
                sayable.say();
                // Referring non-static method using anonymous object
                Sayable sayable2 = new InstanceMethodReference()::saySomething; // You can use anonymous object also
                // Calling interface method
                sayable2.say();
        }
}
```

3) Reference to a Constructor

You can refer a constructor by using the new keyword. Here, we are referring constructor with the help of functional interface.

Syntax

```
ClassName::new
```

Example

```java
interface Messageable{
    Message getMessage(String msg);
}
class Message{
    Message(String msg){
        System.out.print(msg);
    }
}
public class ConstructorReference {
        public static void main(String[] args) {
            Messageable hello = Message::new;
            hello.getMessage("Hello");
        }
    }
```

DefaultMethod:-

Java Default Methods

Java provides a facility to create default methods inside the interface. Methods which are defined inside the interface and tagged with default are known as default methods. These methods are non-abstract methods.

Java Default Method Example

In the following example, Sayable is a functional interface that contains a default and an abstract method. The concept of default method is used to define a method with default implementation. You can override default method also to provide more specific implementation for the method.

Let's see a simple

```java
interface Sayable{
    // Default method
    default void say(){
        System.out.println("Hello, this is default method");
    }
    // Abstract method
    void sayMore(String msg);
}
public class DefaultMethods implements Sayable{
    public void sayMore(String msg){        // implementing abstract method
        System.out.println(msg);
    }
    public static void main(String[] args) {
        DefaultMethods dm = new DefaultMethods();
        dm.say();   // calling default method
        dm.sayMore("Work is worship");  // calling abstract method

    }
}
```

Static Methods inside Java 8 Interface

You can also define static methods inside the interface. Static methods are used to define utility methods. The following example explain, how to implement static method in interface?

```java
interface Sayable{
    // default method
    default void say(){
        System.out.println("Hello, this is default method");
    }
    // Abstract method
    void sayMore(String msg);
    // static method
    static void sayLouder(String msg){
        System.out.println(msg);
    }
}
public class DefaultMethods implements Sayable{
    public void sayMore(String msg){     // implementing abstract method
        System.out.println(msg);
    }
    public static void main(String[] args) {
        DefaultMethods dm = new DefaultMethods();
        dm.say();                        // calling default method
        dm.sayMore("Work is worship");      // calling abstract method
        Sayable.sayLouder("Helloooo...");   // calling static method
    }
}
```

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 allows functional-style operations on the elements. You can use stream by importing java.util.stream package.

- Stream does not store elements. It simply conveys elements from a source such as a data structure, an array, or an I/O channel, through a pipeline of computational operations.
- Stream is functional in nature. Operations performed on a stream does not modify it's source. For example, filtering a Stream obtained from a collection produces a new Stream without the filtered elements, rather than removing elements from the source collection.
- Stream is lazy and evaluates code only when required.
- The elements of a stream are only visited once during the life of a stream. Like an Iterator, a new stream must be generated to revisit the same elements of the source.

You can use stream to filter, collect, print, and convert from one data structure to other etc. In the following examples, we have apply various operations with the help of stream.

the **Stream API** is used to process collections of objects. A stream is a sequence of objects that supports various methods which can be pipelined to produce the desired result.

The features of the Java stream are –

- A stream is not a data structure instead it takes input from the Collections, Arrays, or I/O channels.
- Streams don't change the original data structure, they only provide the result as per the pipelined methods.

In this article, we will explore how we can use the stream API with some examples.

Java 8 Stream API

For processing the objects collections, Java 8 is introduced. A stream is nothing but the objects sequence that give support to the different methods that can be pipelined for producing the result that is required. Before proceeding with this topic further, it is advised to the readers to gain the basic knowledge of Java 8.

Creation of Streams

There are various ways to create the stream instance of the various resources.

Empty Stream

In order to create the empty stream, one must use the empty() method:

1. Syntax: **static** <E> Stream<E> empty()

E: The different types of stream elements.

Return Value: An empty sequential of stream is returned.

Note: When invoking methods that has the stream parameters, an empty stream can be helpful in order to avoid the nullpointer exceptions.

Example

FileName: EmptyStream.java

```java
// important import statement
import java.util.stream.Stream;
import java.util.*;

public class EmptyStream
{

// main method
public static void main(String argvs[])
{
// an empty stream is created here
Stream<String> str = Stream.empty();

// printing elements in Stream
str.forEach(System.out::println);
}
}
```

Output:

```
Nothing to display
```

Using Stream.generate()

Stream generate(Supplier<T> s) returns an infinite sequential unordered stream where every element is produced by the Supplier provided. It is suitable for creating constant streams, streams of the random elements, etc.

Syntax:

```
1. static <E> Stream<E> generate(Supplier<E> sup)
```

Where, stream is an interface and E is the type of stream elements. sup is the Supplier of the elements generated and the value of return is the new infinite sequ

Using Stream.iterate()

The iterate(E, java.util.function.Predicate, java.util.function.UnaryOperator) method allows the iteration of the elements of stream until the mentioned condition. The method returns a sequential ordered Stream generated by the iterative application of the provided next function to the starting element, satisfying the condition hasNext predicate that is being passed as the parameter. The stream gets terminated as soon as the condition hasNext gives a false value.

Syntax:

```
1. static <E> Stream<E> iterate(E st, Predicate<E> hasNext, UnaryOperator<E> next)
```

Parameters: The method has a total of three parameters:

st: it is the starting element,

hasNext: it is the predicate that is applied to the elements for determining whether the stream should terminate or not

next: it is a function that is applied to the previous element in order to produce a new element.

Return value: The method will return a new sequential Stream.

Java Base64 Encode and Decode

Java provides a class Base64 to deal with encryption. You can encrypt and decrypt your data by using provided methods. You need to import java.util.Base64 in your source file to use its methods.

This class provides three different encoders and decoders to encrypt information at each level. You can use these methods at the following levels.

Base64 encoding **allows encoding binary data as text strings for safe transport, at the cost of larger data size.** It is commonly used when there is a need to encode binary data that needs to be stored or transferred via media that are designed to deal with textual data (like transporting images inside an XML, JSON document, etc.).

How to Convert a String to Base64 Encoded String in Java?

In Java, the **java.util.Base64** class provides static methods to encode and decode between binary and base64 formats.

The **encode()** and **decode()** methods are used.

Syntax:

```
String Base64format=
Base64.getEncoder().encodeToString("String".getBytes());
```

Basic Encoding and Decoding

It uses the Base64 alphabet specified by Java in RFC 4648 and RFC 2045 for encoding and decoding operations. The encoder does not add any line separator character. The decoder rejects data that contains characters outside the base64 alphabet.

URL and Filename Encoding and Decoding

It uses the Base64 alphabet specified by Java in RFC 4648 for encoding and decoding operations. The encoder does not add any line separator character. The decoder rejects data that contains characters outside the base64 alphabet.

MIME

It uses the Base64 alphabet as specified in RFC 2045 for encoding and decoding operations. The encoded output must be represented in lines of no more than 76 characters each and uses a carriage return '\r' followed immediately by a linefeed '\n' as the line separator. No line separator is added to the end of the encoded output. All line separators or other characters not found in the base64 alphabet table are ignored in decoding operation.

Example:-

```
import java.util.Base64;

publicclass Base {

    publicstaticvoid main(String[] args) {

        // Getting encoder

        Base64.Encoder encoder = Base64.getUrlEncoder();

        // Encoding URL

        String eStr = encoder.encodeToString("http://www.javatpoint.com/java-tutorial/".getBytes());

        System.out.println("Encoded URL: "+eStr);

        // Getting decoder

        Base64.Decoder decoder = Base64.getUrlDecoder();

        // Decoding URl

        String dStr = new String(decoder.decode(eStr));

        System.out.println("Decoded URL: "+dStr);

    }

}
```

Java for Each loop

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.

---

forEach() Signature in Iterable Interface

```java
default void forEach(Consumer<super T>action)
```

Java 8 forEach() example 1

```java
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));

    }
}
```

Java 9 Try With Resource Enhancement

Java introduced **try-with-resource** feature in Java 7 that helps to close resource automatically after being used.

In other words, we can say that we don't need to close resources (file, connection, network etc) explicitly, try-with-resource close that automatically by using AutoClosable interface.

In Java 7, try-with-resources has a limitation that requires resource to declare locally within its block.

Example Java 7 Resource Declared within resource block

```
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
public class FinalVariable {
    public static void main(String[] args) throws FileNotFoundException {
        try(FileOutputStream fileStream=new FileOutputStream("javatpoint.txt");){
             String greeting = "Welcome to javaTpoint.";
                byte b[] = greeting.getBytes();
                fileStream.write(b);
                System.out.println("File written");
        }catch(Exception e) {
            System.out.println(e);
        }
    }
}
```

Java Type and Repeating Annotations

Java Type Annotations

Java 8 has included two new features repeating and type annotations in its prior annotations topic. In early Java versions, you can apply annotations only to declarations. After releasing of Java SE 8 , annotations can be applied to any type use. It means that annotations can be used anywhere you use

a type. For example, if you want to avoid NullPointerException in your code, you can declare a string variable like this:

```
1. @NonNull String str;
```

Following are the examples of type annotations:

```
1. @NonNull List<String>
1. List<@NonNull String> str
1. Arrays<@NonNegative Integer> sort
1. @Encrypted File file
1. @Open Connection connection
1. void divideInteger(int a, int b) throws @ZeroDivisor ArithmeticException
```

Note - Java created type annotations to support improved analysis of Java programs. It supports way of ensuring stronger type checking.

Java Repeating Annotations

In Java 8 release, Java allows you to repeating annotations in your source code. It is helpful when you want to reuse annotation for the same class. You can repeat an annotation anywhere that you would use a standard annotation.

For compatibility reasons, repeating annotations are stored in a container annotation that is automatically generated by the Java compiler. In order for the compiler to do this, two declarations are required in your code.

Java 8 has included two new features repeating and type annotations in its prior annotations topic. In early Java versions, you can apply annotations only to declarations. After releasing of Java SE 8 , annotations can be applied to any type use. It means that annotations can be used anywhere you use a type. For example, if you want to avoid NullPointerException in your code, you can declare a string variable like this: @NonNull String str;

Java Repeating Annotations

In Java 8 release, Java allows you to repeating annotations in your source code. It is helpful when you want to reuse annotation for the same class. You can repeat an annotation anywhere that you would use a standard annotation.

For compatibility reasons, repeating annotations are stored in a container annotation that is automatically generated by the Java compiler. In order for the compiler to do this, two declarations are required in your code.

1. Declare a repeatable annotation type
2. Declare the containing annotation type

1) Declare a repeatable annotation type

Declaring of repeatable annotation type must be marked with the @Repeatable meta-annotation. In the following example, we have defined a custom @Game repeatable annotation type.

```
@Repeatable(Games.class)
@interfaceGame{
    String name();
    String day();
}
```

The value of the @Repeatable meta-annotation, in parentheses, is the type of the container annotation that the Java compiler generates to store repeating annotations. In the following example, the containing annotation type is Games. So, repeating @Game annotations is stored in an @Games annotation.

2) Declare the containing annotation type

Containing annotation type must have a value element with an array type. The component type of the array type must be the repeatable annotation type. In the following example, we are declaring Games containing annotation type:

```
@interfaceGames{
    Game[] value();  }
```

Note - Compiler will throw a compile-time error, if you apply the same annotation to a declaration without first declaring it as repeatable.

Example:-

```
class Meerut
{
void print()
{
System.out.println("hello");
}
}
class Charlie extends Meerut
{
@Override void print()
{ System.out.println("hii");
}
  @Deprecated
public static void main(String args[])
{
Charlie c=new Charlie();
c.print();
}
}
```

Java 9 Module System

Java Module System is a major change in Java 9 version. Java added this feature to collect Java packages and code into a single unit called *module*.

In earlier versions of Java, there was no concept of module to create modular Java applications, that why size of application increased and difficult to move around. Even JDK itself was too heavy in size, in Java 8, **rt.jar** file size is around 64MB.

To deal with situation, **Java 9 restructured JDK into set of modules** so that we can use only required module for our project.

Apart from JDK, Java also allows us to create our own modules so that we can develop module based application.

The module system includes various tools and options that are given below.

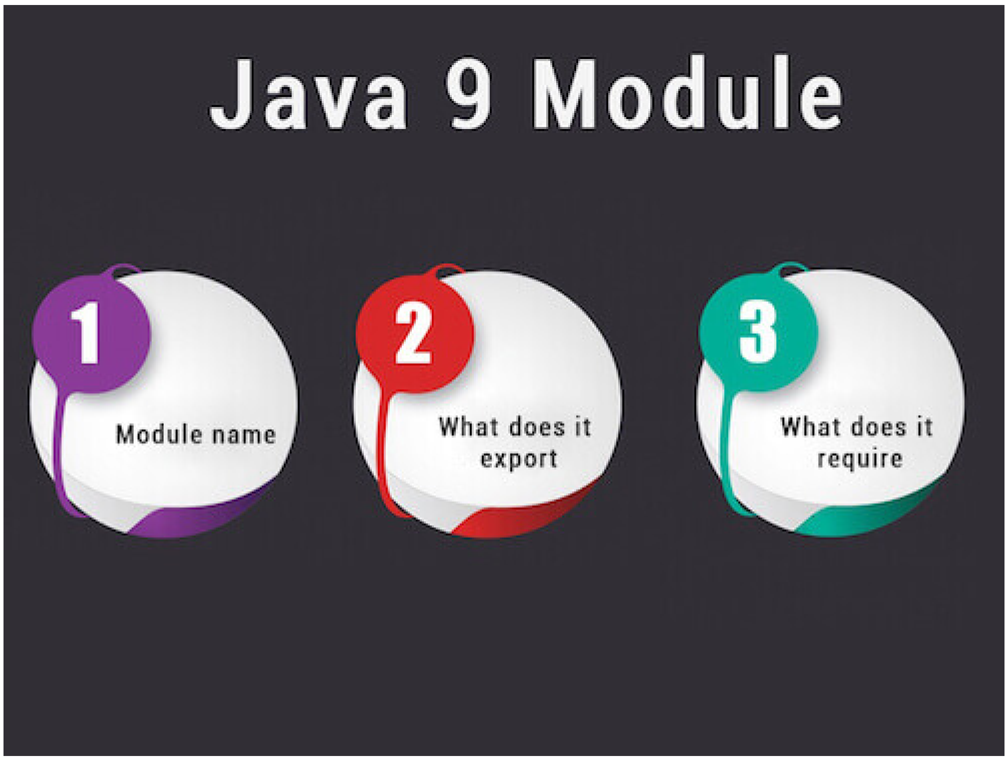
- Includes various options to the Java tools **javac, jlink and java** where we can specify module paths that locates to the location of module.

- Modular JAR file is introduced. This JAR contains module-info.class file in its root folder.
- JMOD format is introduced, which is a packaging format similar to JAR except it can include native code and configuration files.
- The JDK and JRE both are reconstructed to accommodate modules. It improves performance, security and maintainability.
- Java defines a new URI scheme for naming modules, classes and resources.

Java 9 Module

Module is a collection of Java programs or softwares. To describe a module, a Java file **module-info.java** is required. This file also known as module descriptor and defines the following

- Module name
- What does it export
- What does it require



Module Name

It is a name of module and should follow the reverse-domain-pattern. Like we name packages, e.g. com.javatpoint.

How to create Java module

Creating Java module required the following steps.

- Create a directory structure
- Create a module declarator
- Java source code

Create a Directory Structure

To create module, it is recommended to follow given directory structure, it is same as reverse-domain-pattern, we do to create packages / project-structure in Java.

Note: The name of the directory containing a module's sources should be equal to the name of the module, e.g. com.javatpoint.

```
src
 └── com.javatpoint
     ├── com
     │   └── javatpoint
     │       └── Hello.java
     └── module-info.java
```

Create a file **module-info.java**, inside this file, declare a module by using **module** identifier and provide module name same as the directory name that contains it. In our case, our directory name is com.javatpoint.

```
module com.javatpoint{

}
```

Leave module body empty, if it does not has any module dependency. Save this file inside **src/com.javatpoint** with **module-info.java** name.

Java Source Code

Now, create a Java file to compile and execute module. In our example, we have a **Hello.java** file that contains the following code.

```java
class Hello{
    public static void main(String[] args){
        System.out.println("Hello from the Java module");
    }
}
```

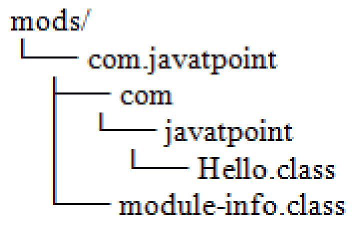
Save this file inside **src/com.javatpoint/com/javatpoint/** with **Hello.java** name.

Compile Java Module

To compile the module use the following command.

```
javac -d mods --module-source-path src/ --module com.javatpoint
```

After compiling, it will create a new directory that contains the following structure.



```
mods/
└── com.javatpoint
    ├── com
    │   └── javatpoint
    │       └── Hello.class
    └── module-info.class
```

Now, we have a compiled module that can be just run.

Run Module

To run the compiled module, use the following command.

```
java --module-path mods/ --module com.javatpoint/com.javatpoint.Hello
```

Output:

```
Hello from the Java module
```

Well, we have successfully created, compiled and executed Java module.

Look inside compiled Module Descriptor

To see the compiled module descriptor use the following command.

```
javap mods/com.javatpoint/module-info.class
```

This command will show the following code to the console.

```
Compiled from "module-info.java"
module com.javatpoint {
  requires java.base;
}
```

See, we created an empty module but it contains a **java.base** module. Why? Because all Java modules are linked to java.base module and it is default module.

Diamond operator for Anonymous Inner Class with Examples in Java

Diamond Operator: Diamond operator was introduced in Java 7 as a new feature.The main purpose of the diamond operator is to simplify the use of generics when creating an object. It avoids unchecked warnings in a program and makes the program more readable. The diamond operator could not be used with Anonymous inner classes in JDK 7. In JDK 9, it can be used with the anonymous

class as well to simplify code and improves readability. Before JDK 7, we have to create an object with Generic type on both side of the expression like:
With the help of Diamond operator, we can create an object without mentioning the generic type on the right hand side of the expression. But the problem is it will only work with normal classes.

```
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);
    }
}
```

Java Anonymous inner class

Java anonymous inner class is an inner class without a name and for which only a single object is created. An anonymous inner class can be useful when making an instance of an object with certain "extras" such as overloading methods of a class or interface, without having to actually subclass a class.

In simple words, a class that has no name is known as an anonymous inner class in Java. It should be used if you have to override a method of class or interface. Java Anonymous inner class can be created in two ways:

1. Class (may be abstract or concrete).
2. Interface

Java anonymous inner class example using class

TestAnonymousInner.java

```
abstract class Person{
  abstract void eat();
}
class TestAnonymousInner{
 public static void main(String args[]){
  Person p=new Person(){
  void eat(){System.out.println("nice fruits");}
  };
  p.eat();
 }
}
```

Output: ice fruits

Internal working of given code

```
Person p=new Person(){
void eat(){System.out.println("nice fruits");}
};
```

1. A class is created, but its name is decided by the compiler, which extends the Person class and provides the implementation of the eat() method.
2. An object of the Anonymous class is created that is referred to by 'p,' a reference variable of Person type.

Internal class generated by the compiler

```
import java.io.PrintStream;
static class TestAnonymousInner$1 extends Person
{
   TestAnonymousInner$1(){}
   void eat()
    {
        System.out.println("nice fruits");
    }
}
```

Java anonymous inner class example using interface

```
interface Eatable{
 void eat();
}
class TestAnnonymousInner1{
 public static void main(String args[]){
 Eatable e=new Eatable(){
  public void eat(){System.out.println("nice fruits");}
 };
 e.eat();
 }
}
```

Test it Now

Output:

```
nice fruits
```

Internal working of given code

It performs two main tasks behind this code:

```
Eatable p=new Eatable(){
void eat(){System.out.println("nice fruits");}
};
```

1. A class is created, but its name is decided by the compiler, which implements the Eatable interface and provides the implementation of the eat() method.
2. An object of the Anonymous class is created that is referred to by 'p', a reference variable of the Eatable type.

Internal class generated by the compiler

```
import java.io.PrintStream;
static class TestAnonymousInner1$1 implements Eatable
{
TestAnonymousInner1$1(){}
void eat(){System.out.println("nice fruits");}
}
```

Switch Expressions

Like all expressions, switch expressions evaluate to a single value and can be used in statements. They may contain "case L ->" labels that eliminate the need
for break statements to prevent fall through. You can use a yield statement to specify the value of a switch expression.
The yield statement makes it easier for you to differentiate between switch statements and switch expressions. A switch statement, but not a switch expression, can be the target of a break statement. Conversely, a switch expression, but not a switch statement, can be the target of a yield statement.
Unlike switch statements, the cases of switch expressions must be *exhaustive*, which means that for all possible values, there must be a matching switch label.
Thus, switch expressions normally require a default clause. However,
for enum switch expressions that cover all known constants, the compiler inserts an implicit default clause.

In addition, a switch expression must either complete normally with a value or complete abruptly by throwing an exception. For example, the following code doesn't compile because the switch labeled rule doesn't contain a yield statement:
Copy
```
int i = switch (day) {
    case MONDAY -> {
        System.out.println("Monday");
        // ERROR! Block doesn't contain a yield statement
    }
    default -> 1;
};
```

Example of Switch Expression

```
class Swit

{

public enum Day { SUNDAY, MONDAY, TUESDAY,WEDNESDAY,
THURSDAY, FRIDAY, SATURDAY; }

public static void main(String args[])

{

  int numLetters = 0;

         Day day = Day.WEDNESDAY;

    numLetters = switch (day) {

        case MONDAY:

        case FRIDAY:

        case SUNDAY:

            System.out.println(6);

            yield 6;

        case TUESDAY:

            System.out.println(7);

            yield 7;

        case THURSDAY:

        case SATURDAY:

            System.out.println(8);
            yield 8;
        case WEDNESDAY:
            System.out.println(9);
            yield 9;
        default:
            throw new IllegalStateException("Invalid day: " + day);
    };
    System.out.println(numLetters);
}
}
```

2. The *yield* Keyword

The *yield* keyword lets us exit a *switch* expression by returning a value that becomes the value of the *switch* expression.

This means we can assign the value of a *switch* expression to a variable.

Lastly, by using *yield* in a *switch* expression, we get an implicit check that we're covering our cases, which makes our code more robust.

Let's look at some examples.

yield with Arrow Operator

To start, let's say we have the following *enum* and *switch* statement:

```
public enum Number {
    ONE, TWO, THREE, FOUR;
}

String message;
switch (number) {
    case ONE:
        message = "Got a 1";
        break;
    case TWO:
        message = "Got a 2";
        break;
    default:
        message = "More than 2";
}
```
Copy

Let's convert this to a *switch* expression and use the *yield* keyword along with the arrow operator:

```
String message = switch (number) {
    case ONE -> {
        yield "Got a 1";
    }
    case TWO -> {
        yield "Got a 2";
    }
    default -> {
        yield "More than 2";
    }
};
```
Copy

yield sets the value of the *switch* expression depending on the value of *number*.

2.2. *yield* with Colon Delimiter

We can also create a *switch* expression using *yield* with the colon delimiter:

```
String message = switch (number) {
    case ONE:
        yield "Got a 1";
    case TWO:
        yield "Got a 2";
    default:
        yield "More than 2";
};
```

This code behaves the same as in the previous section. But the arrow operator is clearer and also less prone to forgetting *yield* (or *break*) statements.

We should note that we can't mix colon and arrow delimiters within the same *switch* expression.

TEXT BLOCK

Text blocks start with a "''" (three double-quote marks) followed by optional whitespaces and a newline. The most simple example looks like this:

```
String example = """
     Example text""";
```

Note that the result type of a text block is still a *String.* Text blocks just provide us with another way to write *String* literals in our source code.

A text block is an alternative form of Java string representation that can be used anywhere a traditional double-quoted string literal can be used. Text blocks begin with a "'' (3 double-quote marks) observed through non-obligatory whitespaces and a newline. For example:

```
// Using a literal string

String text1 = "Geeks For Geeks";
```

```
// Using a text block

String text2 = """

                Geeks For Geeks""";
```

Example:-Text Block

```
class New
{
public static void main(String args[])
{
String s="""
hiiiii this is a new feature of java 123c""";
System.out.println(s);
}
}
```

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 variable

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.

Local Variable Type Inference, or `type inference`, refers to the automatic detection of a data type by the compiler based on the initialized value.

Example:

```
class Renu

{

public static void main(String args[])

{


var name = "Welcome to tutorialspoint.com";

var x=432;

System.out.println(name+" "+x);

}

}
```

Java 10 further simplifies and reduces code by introducing the var keyword, maintaining type safety. The same ArrayList can now be declared as:

```
var lists=new ArrayList<String>();
```

The compiler infers the type from the right-hand

#How Compiler Interprets var Types

When encountering a local variable with `var`, the compiler checks the right-hand side (initializer) to determine its type and assigns this type to the variable.

Let’s explore some examples of using var.

#Basic Type Inference Example - var Keyword

In the below example, the local string is declared and initialized with string constant literal value.

Another local variable is declared and used to store the result of the lowercase of a string.

Type is not declared. The compiler automatically inferred the type from its value.

```
public class LocalVariableTypeInferenceHelloWorld {
    public static void main(String[] args) {
        var str = "THIS IS STRING";
        var lowerCaseString = str.toLowerCase();
        System.out.println(lowerCaseString);
        System.out.println(lowerCaseString.getClass().getTypeName());
    }
}
```

#For Loop local variable type inference example in java10

Declared `index` variables in for loop as like below.

```
public class Test {
    public static void main(String[] args) {
        for (var i = 1; i <= 10; i++) {
            var temp= i* 2; // equals to Integer temp=i* 2;
            System.out.println(temp);
        }
    }
}
```

#local variable type Inference Compilation Errors

Local variables also have a lot of limitations and restrictions on the usage of this.

The below cases all give compilation errors.

1. **No Local variable without an initializer**

```
var str;
```

Here local variable declares, but not initialized, it gives compilation errors like

1. **No variable initialized with a null value**

```
var str=null;
```

if var variable initialized with a null value, Compiler gives error - *Cannot infer type for local variable initialized to 'null'*.

1. **No Multiple or compound variables declaration**

multiple local variables declarations are not allowed

```
var m=5,n=2,p=3;  // not allowed
int m=5,n=2,p=3;  //  allowed
```

if we declare multiple local variables, It gives *var’ is not allowed in a compound declaration*‘.

1. **No Local var array initializer**

An array declared like below is not allowed for local variables.

```
var arrayDemo = { 61 , 14 };
```

the above line of code gives error *Array initializer needs an explicit target-type*

1. **Not class instance variables**

instance variables or class variables declared with local variables are not allowed.

```
public class ClassLocalDemo {
 var memberVariable=10;
 public static void main(String[] args) {
 }
}
```

1. **No Method arguments/parameters**

local variables do not declare in the method signature.

```
public class ClassLocalDemo {
 public static void main(String[] args) {
 }
 public static void method(var parame) {
 }
}
```

1. **No var in Method return type**

method return type should not be var word and throws a compilation error

```
public static var method() {
 }
```

1. **No var declaration in a catch block**

var is not allowed catch block as like below

```
public class Test {
    public static void main(String[] args) {
        try {
            // code
        } catch (var e) {
        }
    }
```

RECORD

Java records were introduced as a preview feature in Java 14 [JEP-359] and finalized in Java 16 [JEP-395]. **A *record*, in Java, acts as a transparent carrier for immutable data.** Conceptually, records can be thought of as tuples that are already available via 3rd party libraries.

Though, records are built-in type in Java so they provide a more extended use and compatibility with other features in Java such as pattern matching with *instanceof* and switch case.

What is a Record?

Like enum, a record is also a special class type in Java. **Records are intended to be used in places where a class is created only to act as a plain data carrier**.

The important **difference between 'class' and 'record' is that a *record* aims to eliminate all the boilerplate code needed to set and get the data from the instance**. Records transfer this responsibility to the Java compiler, which generates the constructor, field getters, hashCode() and equals() as well toString() methods.

Note that we can override any of the default provided above methods in record definition to implement custom behavior.

The Class of a Record

A record is class declared with the record keyword instead of the class keyword. Let us declare the following record.

```
public record Point(int x, int y) {}
```

The class that the compiler creates for you when you create a record is final.

This class extends the java.lang.Record class. So your record cannot extend any class.

A record can implement any number of interfaces.

Declaring the Components of a Record

The block that immediately follows the name of the record is (int x, int y). It declares the *components* of the record named Point. For each component of a record, the compiler creates a private final field with the same name as this component. You can have any number of components declared in a record.

In this example, the compiler creates two private final fields of type int: x and y, corresponding to the two components you have declared.

Along with these fields, the compiler generates one *accessor* for each component. This accessor is a method that has the same name of the component, and returns its value. In the case of this Point record, the two generated methods are the following.

```
public int x() {
    return this.x;
}

public int y() {
    return this.y;
}
```

Copy

If this implementation works for your application, then you do not need to add anything. You may define your own accessor methods though. It may be useful in the case where you need to return a defensive copy of a particular field.

The last elements generated for you by the compiler are overrides of the toString(), equals() and hashCode() methods from the Object class. You may define your own overrides of these methods if you need.

Things you Cannot Add to a Record

There are three things that you cannot add to a record:

1. You cannot declare any instance field in a record. You cannot add any instance field that would not correspond to a component.
2. You cannot define any field initializer.
3. You cannot add any instance initializer.

You can create static fields with initializers and static initializers.

Constructing a Record with its Canonical Constructor

The compiler also creates a constructor for you, called the *canonical constructor*. This constructor takes the components of your record as arguments and copies their values to the fields of the record class.

There are situations where you need to override this default behavior. Let us examine two use cases:

1. You need to validate the state of your record
2. You need to make a defensive copy of a mutable component.

Using the Compact Constructor

You can use two different syntax to redefine the canonical constructor of a record. You can use a compact constructor or the canonical constructor itself.

Suppose you have the following record.

```
public record Range(int start, int end) {}
```

Copy

For a record of that name, one could expect that the end is greater that the start. You can add a validation rule by writing the compact constructor in your record.

```
public record Range(int start, int end) {


    public Range {
        if (end <= start) {
            throw new IllegalArgumentException("End cannot be lesser 
than start");
        }
    }
}
```

Copy

The compact canonical constructor does not need to declare its block of parameters.

Note that if you choose this syntax, you cannot directly assign the record's fields, for example with this.start = start - that is done for you by code added by the compiler. But you can assign new values to the parameters, which leads to the same result because the compiler-generated code will then assign these new values to the fields.

```
public Range {
    // set negative start and end to 0
    // by reassigning the compact constructor's
    // implicit parameters
    if (start < 0)
        start = 0;
    if (end < 0)
        end = 0;
}
```

Copy

Using the Canonical Constructor

If you prefer the non-compact form, for example because you prefer not to reassign parameters, you can define the canonical constructor yourself, as in the following example.

```
public record Range(int start, int end) {

    public Range(int start, int end) {
        if (end <= start) {
            throw new IllegalArgumentException("End cannot be lesser 
than start");
        }
        if (start < 0) {
            this.start = 0;
        } else {
            this.start = start;
        }
        if (end > 100) {
            this.end = 10;
        } else {
            this.end = end;
        }
    }
}
```
Copy

In this case the constructor you write needs to assign values to the fields of your record.

If the components of your record are not immutable, you should consider making defensive copies of them in both the canonical constructor and the accessors.

Defining any Constructor

You can also add any constructor to a record, as long as this constructor calls the canonical constructor of your record. The syntax is the same as the classic syntax that calls a constructor with another constructor. As for any class, the call to `this()` must be the first statement of your constructor.

Let us examine the following `State` record. It is defined on three components:

1. the name of this state
2. the name of the capital of this state
3. a list of city names, that may be empty.

We need to store a defensive copy of the list of cities, to ensure that it will not be modified from the outside of this record. This can be done by redefining the canonical constructor with a compact form that reassigns the parameter to the defensive copy.

Having a constructor that does not take any city is useful in your application. This can be another constructor, that only takes the state name and the capital city name. This second constructor must call the canonical constructor.

Then, instead of passing a list of cities, you can pass the cities as a vararg. To do that, you can create a third constructor, that must call the canonical constructor with the proper list.

```
public record State(String name, String capitalCity, List<String>
cities) {

    public State {
        // List.copyOf returns an unmodifiable copy,
        // so the list assigned to `cities` can't change anymore
        cities = List.copyOf(cities);
    }
    public State(String name, String capitalCity) {
        this(name, capitalCity, List.of());
    }

    public State(String name, String capitalCity, String... cities) {
        this(name, capitalCity, List.of(cities));
    }

}
```
Copy

Note that the List.copyOf() method does not accept null values in the collection it gets as an argument.

Example:Record

```
public record Range(int start, int end) {

    public Range(int start, int end) {
        if (end <= start) {
            throw new IllegalArgumentException("End cannot be lesser than start");
        }
        if (start < 0) {
            this.start = 0;
System.out.println(start+" "+end);
        } else {
            this.start = start;
        }
        if (end > 100) {
            this.end = 10;
System.out.println(start+" "+end);

        } else {
            this.end = end;
System.out.println(start+" "+end);
```

```
        }
    }
public static void main(String args[])
{
Range r=new Range(2,78);

}
}
```

Sealed Class in Java

In programming, security and control flow are the two major concerns that must be considered while developing an application. There are various controlling features such as the use of final and protected keyword restricts the user to access variables and methods. Java 15 introduces a new **preview feature** that allows us to control the inheritance. In this section, we will discuss the **preview feature, the concept of sealed class, and interface** with proper examples.

Java Sealed Class

Java 15 introduced the concept of **sealed** classes. It is a preview feature. Java sealed classes and interfaces restrict that which classes and interfaces may extend or implement them.

In other words, we can say that the class that cannot be inherited but can be instantiated is known as the sealed class. It allows classes and interfaces to have more control over their permitted subtypes. It is useful both for general domain modeling and for building a more secure platform for libraries.

Note that the concept of sealed classes is a **preview feature,** not a permanent feature.

Steps to Create a Sealed Class

- Define the class that you want to make a seal.
- Add the “sealed” keyword to the class and specify which classes are permitted to inherit it by using the “permits” keyword.

Example

```
sealed class Human permits Manish, Vartika, Anjali
{
   public void printName()
   {
       System.out.println("Default");
   }
}
non-sealed class Manish extends Human
{
   public void printName()
   {
       System.out.println("Manish Sharma");
   }
}
sealed class Vartika extends Human
{
   public void printName()
   {
       System.out.println("Vartika Dadheech");
   }
}
final class Anjali extends Human
{
   public void printName()
   {
       System.out.println("Anjali Sharma");
   }
}
```

Explanation of the above Example:

- *Human* is the parent class of *Manish*, *Vartika,* and *Anjali*. It is a sealed class so; other classes cannot inherit it.
- *Manish*, *Vartika*, and *Anjali* are child classes of the *Human* class, and it is necessary to make them either *sealed*, *non-sealed*, or *final*. Child classes of a sealed class must be sealed, non-sealed, or final.
- If any class other than *Manish*, *Vartika*, and *Anjali* tries to inherit from the *Human* class, it will cause a compiler error.

Advantages of Sealed Class and Interface

- It allows permission to the subclasses that can extend the sealed superclass.
- It makes superclass broadly accessible but not broadly extensible.
- It allows compilers to enforce the type system on the users of the class.
- Developer of a superclass gets control over the subclasses. Hence, they can define methods in a more restricted way.

Defining a Sealed Class

The declaration of a sealed class is not much complicated. If we want to declare a class as sealed, add a **sealed** modifier to its declaration. After the class declaration and extends and implements clause, add **permits** clause. The clause denotes the classes that may extend the sealed class.

It presents the following modifiers and clauses:

- **sealed:** It can only be extended by its permitted subclasses.
- **non-sealed:** It can be extended by unknown subclasses; a sealed class cannot prevent its permitted subclasses from doing this.
- **permits:** It allows the subclass to inherit and extend.
- **final:** The permitted subclass must be final because it prevents further extensions.