UNIT-3

Course Outcome:

CO 3: Apply new java features to build java programs.

Syllabus:

Java New Features: Java New Features: Functional Interfaces, Lambda Expression, Method References, Stream API, Default Methods, Static Method, Base64 Encode and Decode, ForEach Method, Try-withresources, Type Annotations, Repeating Annotations, Java Module System, Diamond Syntax with Inner Anonymous Class, Local Variable Type Inference, Switch Expressions, Yield Keyword, Text Blocks, Records, Sealed Classes**.**

Lecture Delivery Plan:

Lecture-19

19.1 Functional Interfaces

19.2 Lambda Expression

Lecture-20

20.1 Method References

20.2 Stream API,

20.3 Default Methods

Lecture-21

21.1 Static Method,

21.2 Base64 Encode and Decode

Lecture-22

22.1 ForEach Method,

22.2 Try-with resources

Lecture-23

23.1 Type Annotations,

23.2 Repeating Annotations

23.3 Java Module System

Lecture-24

24.1 Diamond Syntax with Inner Anonymous Class

24.2 Local Variable Type Inference,

Lecture-25

25.1 Switch Expressions

25.2 Yield Keyword,

Lecture-26

26.1 Text Blocks

26.2 Records

26.3 Sealed Classes

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 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 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.

Java 8 Stream

Java provides a new additional package in Java 8 called java.util.stream.

This package consists of class 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.

4.1 Streams API

The newly added Stream API (**java.util.stream**) introduces real-world functional-style programming into the Java. This is by far the most comprehensive addition to Java library intended to make Java developers significantly more productive by allowing them to write effective, clean, and concise code.

Stream API makes collections processing greatly simplified (but it is not limited to Java collections only as we will see later). Let us take start off with simple

class called Task.

```
public class Streams  {
    private enum Status {

        OPEN, CLOSED

    };


    private static final class Task {
        private final Status status;
        private final Integer points;


        Task( final Status status, final Integer points ) { this.status
            = status;

            this.points = points;

        }


        public Integer getPoints() {
            return points;

        }


        public Status getStatus() {
            return status;

        }
```

Task has some notion of points (or pseudo-complexity) and can be either **OPEN** or **CLOSED**. And then let us introduce a small collection of tasks to play with.

```
final Collection< Task > tasks = Arrays.asList( new
    Task( Status.OPEN, 5 ),

    new Task( Status.OPEN, 13 ), new
    Task( Status.CLOSED, 8 )
```

The first question we are going to address is how many points in total all **OPEN** tasks have? Up to Java 8, the usual solution for it would be some sort of **foreach** iteration. But in Java 8 the answers is streams: a sequence of elements supporting sequential and parallel aggregate operations.

```
// Calculate total points of all active tasks using sum()
final long totalPointsOfOpenTasks = tasks

    .stream()

    .filter( task -> task.getStatus() == Status.OPEN )

    .mapToInt( Task::getPoints )
```

And the output on the console looks like that:

```
Total points: 18
```

There are a couple of things going on here. Firstly, the tasks collection is converted to its stream representation. Then, the **filter** operation on stream filters out all **CLOSED** tasks. On next step, the **mapToInt** operation converts the

stream of **Tasks** to the stream of **Integers** using **Task::getPoints** method of the each task instance. And lastly, all points are summed up using **sum** method, producing the final result.

Before moving on to the next examples, there are some notes to keep in mind about streams (more details here). Stream operations are divided into intermediate and terminal operations.

Intermediate operations return a new stream. They are always lazy, executing an intermediate operation such as **filter** does not actually perform any filtering, but instead creates a new stream that, when traversed, contains the elements of the initial stream that match the given predicate

Terminal operations, such as **forEach** or **sum**, may traverse the stream to produce a result or a side-effect. After the terminal operation is performed, the stream pipeline is considered consumed, and can no longer be used. In almost all cases, terminal operations are eager, completing their traversal of the underlying data source.

Yet another value proposition of the streams is out-of-the box support of parallel processing. Let us take a look on this example, which does sums the points of all the tasks.

```
// Calculate total points of all tasks final
double totalPoints = tasks

   .stream()

   .parallel()

   .map( task -> task.getPoints() ) // or map( Task::getPoints )
```

It is very similar to the first example except the fact that we try to process all the tasks in **parallel** and calculate the final result using **reduce** method.

Here is the console output:

```
Total points (all tasks): 26.0
```

Often, there is a need to performing a grouping of the collection elements by some criteria. Streams can help with that as well as an example below demonstrates.

```
// Group tasks by their status

final Map< Status, List< Task > > map = tasks

    .stream()
```

The console output of this example looks like that:

```
{CLOSED=[[CLOSED, 8]], OPEN=[[OPEN, 5], [OPEN, 13]]}
```

To finish up with the tasks example, let us calculate the overall percentage (or weight) of each task across the whole collection, based on its points.

```
// Calculate the weight of each tasks (as percent of total points) final
Collection< String > result = tasks

    .stream()                                          // Stream< String >

    .mapToInt( Task::getPoints )                       // IntStream

    .asLongStream()                                    // LongStream

    .mapToDouble( points -> points / totalPoints )     // DoubleStream
```

```
    .collect( Collectors.toList() );                    // List< String >

System.out.println( result );
```

The console output is just here:

```
[19%, 50%, 30%]
```

And lastly, as we mentioned before, the Stream API is not only about Java collections. The typical I/O operations like reading the text file line by line is a very good candidate to benefit from stream processing. Here is a small example to confirm that.

```
final Path path = new File( filename ).toPath();

try( Stream< String > lines = Files.lines( path, StandardCharsets.UTF_8 ) ) { lines.onClose(
    () -> System.out.println("Done!") ).forEach( System.out::println );
```

The **onClose** method called on the stream returns an equivalent stream with an additional close handler. Close handlers are run when the **close()** method is called on the stream.

Stream API together with Lambdas and Method References baked by Interface's Default and Static Methods is the Java 8 response to the modern paradigms in software development. For more details, please refer to official documentation.

4.2 Date/Time API (JSR 310)

Java 8 makes one more take on date and time management by delivering New Date-Time API (JSR 310). Date and time manip- ulation is being one of the worst pain points for Java developers. The standard **java.util.Date** followed by **java.util.Calendar** hasn't improved the situation at all (arguably, made it even more confusing).

That is how Joda-Time was born: the great alternative date/time API for Java. The Java 8's New Date-Time API (JSR 310) was heavily influenced by Joda-Time and took the best of it. The new **java.time** package contains all the classes for date, time, date/time, time zones, instants, duration, and clocks manipulation. In the design of the API the immutability has been taken into account very seriously: no change allowed (the tough lesson learnt from * java.util.Calendar*). If the modification is required, the new instance of respective class will be returned.

Let us take a look on key classes and examples of their usages. The first class is **Clock** which provides access to the current instant, date and time using a time-zone. **Clock** can be used instead of **System.currentTimeMillis()** and **TimeZone.getDefault()**.

```
// Get the system clock as UTC offset final
Clock clock = Clock.systemUTC();
System.out.println( clock.instant() );
System.out.println( clock.millis() );
```

The sample output on a console:

```
2014-04-12T15:19:29.282Z

1397315969360
```

Other new classes we are going to look at are **LocaleDate** and **LocalTime**. **LocaleDate** holds only the date part without a time-zone in the ISO-8601 calendar system. Respectively, **LocaleTime** holds only the time part without time-zone in the ISO-8601 calendar system. Both **LocaleDate** and **LocaleTime** could be created from **Clock**.

```
// Get the local date and local time final
LocalDate date = LocalDate.now();

final LocalDate dateFromClock = LocalDate.now( clock );


System.out.println( date );
System.out.println( dateFromClock );
```

```
final LocalTime timeFromClock = LocalTime.now( clock );


System.out.println( time );
System.out.println( timeFromClock );
```

The sample output on a console:

```
2014-04-12

2014-04-12

11:25:54.568

15:25:54.568
```

The **LocalDateTime** combines together **LocaleDate** and **LocalTime** and holds a date with time but without a time-zone in the ISO-8601 calendar system. A quick example is shown below.

```
// Get the local date/time

final LocalDateTime datetime = LocalDateTime.now();

final LocalDateTime datetimeFromClock = LocalDateTime.now( clock );
```

The sample output on a console:

```
2014-04-12T11:37:52.309

2014-04-12T15:37:52.309
```

If case you need a date/time for particular timezone, the **ZonedDateTime** is here to help. It holds a date with time and with a time-zone in the ISO-8601 calendar system. Here are a couple of examples for different timezones.

```
// Get the zoned date/time

final ZonedDateTime zonedDatetime = ZonedDateTime.now();

final ZonedDateTime zonedDatetimeFromClock = ZonedDateTime.now( clock );

final ZonedDateTime zonedDatetimeFromZone = ZonedDateTime.now( ZoneId.of( "America/ ↩
    Los_Angeles" ) );
```

The sample output on a console:

```
2014-04-12T11:47:01.017-04:00[America/New_York]

2014-04-12T15:47:01.017Z
```

And finally, let us take a look on **Duration** class: an amount of time in terms of seconds and nanoseconds. It makes very easy to compute the different between two dates. Let us take a look on that.

```
// Get duration between two dates

final LocalDateTime from = LocalDateTime.of( 2014, Month.APRIL, 16, 0, 0, 0 );

final LocalDateTime to = LocalDateTime.of( 2015, Month.APRIL, 16, 23, 59, 59 );


final Duration duration = Duration.between( from, to );
```

The example above computes the duration (in days and hours) between two dates, **16 April 2014** and **16 April 2015**. Here is the sample output on a console:

```
Duration in days: 365
Duration in hours: 8783
```

he overall impression about Java 8's new date/time API is very, very positive. Partially, because of the battle-proved foundation it is built upon (Joda-Time), partially because this time it was finally tackled seriously and developer voices have been heard. For more details please refer to official documentation.

2.1 Interface's Default and Static Methods

Java 8 extends interface declarations with two new concepts: default and static methods. Default methods make interfaces somewhat similar to traits but serve a bit different goal. They allow adding new methods to existing interfaces without breaking the binary compatibility with the code written for older versions of those interfaces.

The difference between default methods and abstract methods is that abstract methods are required to be implemented. But default methods are not. Instead, each interface must provide so called default implementation and all the implementers will inherit it by default (with a possibility to override this default implementation if needed). Let us take a look on example below.

```
private interface Defaulable {

    // Interfaces now allow default methods, the implementer may or

    // may not implement (override) them.
    default String notRequired() {

        return "Default implementation";

    }
```

```
private static class OverridableImpl implements Defaulable { @Override

    public String notRequired() {

        return "Overridden implementation";

    }
```

The interface **Defaulable** declares a default method **notRequired()** using keyword **default** as part of the method definition. One of the classes, **DefaultableImpl**, implements this interface leaving the default method implementation as-is. Another one, **OverridableImpl** , overrides the default implementation and provides its own.

Another interesting feature delivered by Java 8 is that interfaces can declare (and provide implementation) of static methods. Here is an example.

```
private interface DefaulableFactory {

    // Interfaces now allow static methods

    static Defaulable create( Supplier< Defaulable > supplier ) { return
        supplier.get();
```

The small code snippet below glues together the default methods and static methods from the examples above.

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

    Defaulable defaulable = DefaulableFactory.create( DefaultableImpl::new ); System.out.println(
    defaulable.notRequired() );


    defaulable = DefaulableFactory.create( OverridableImpl::new );
    System.out.println( defaulable.notRequired() );
```

The console output of this program looks like that:

```
Default implementation
Overridden implementation
```

Default methods implementation on JVM is very efficient and is supported by the byte code instructions for method invocation. Default methods allowed existing Java interfaces to evolve without breaking the compilation process. The good examples are the plethora of methods added to **java.util.Collection** interface: **stream()**, **parallelStream()**, **forEach()**, **removeIf()**, . . .

Though being powerful, default methods should be used with a caution: before declaring method as default it is better to think twice if it is really needed as it may cause ambiguity and compilation errors in complex hierarchies. For more details please refer to official documentation.

2.2 Method References

Method references provide the useful syntax to refer directly to exiting methods or

constructors of Java classes or objects (in- stances). With conjunction of Lambdas expressions, method references make the language constructs look compact and concise, leaving off boilerplate.

Below, considering the class **Car** as an example of different method definitions, let us distinguish four supported types of method references.

```
public static class Car {

    public static Car create( final Supplier< Car > supplier ) { return
        supplier.get();

    }
```

```
        System.out.println( "Collided " + car.toString() );
    }

    public void follow( final Car another ) {
        System.out.println( "Following the " + another.toString() );
    }

    public void repair() {
        System.out.println( "Repaired " + this.toString() );
    }
}
```

The first type of method references is constructor reference with the syntax **Class::new** or alternatively, for generics, **Class< T >::new**. Please notice that the constructor has no arguments.

```
final Car car = Car.create( Car::new );
final List< Car > cars = Arrays.asList( car );
```

The second type is reference to static method with the syntax **Class::static_method**. Please notice that the method accepts exactly one parameter of type **Car**.

```
cars.forEach( Car::collide );
```

The third type is reference to instance method of arbitrary object of specific type with the syntax **Class::method**. Please notice, no arguments are accepted by the method.

```
cars.forEach( Car::repair );
```

And the last, fourth type is reference to instance method of particular class instance the syntax **instance::method**. Please notice that method accepts exactly one parameter of type **Car**.

```
final Car police = Car.create( Car::new ); cars.forEach(
police::follow );
```

Running all those examples as a Java program produces following output on a

console (the actual **Car** instances might be different):

```
Collided com.javacodegeeks.java8.method.references.MethodReferences$Car@7a81197d
Repaired com.javacodegeeks.java8.method.references.MethodReferences$Car@7a81197d
Following the com.javacodegeeks.java8.method.references.MethodReferences$Car@7a81197d
```

For more examples and details on method references, please refer to official documentation.

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.

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.

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.

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

```
1.  interface Sayable{
2.  // Default method
3.  default void say(){
4.  System.out.println("Hello, this is default method");
5.  }
6.  // Abstract method
7.  void sayMore(String msg);
8.  }
9.  public class DefaultMethods implements Sayable{
10. public void sayMore(String msg){      // implementing abstract method
11. System.out.println(msg);
12. }
13. public static void main(String[] args) {
14. DefaultMethods dm = new DefaultMethods();
15. dm.say();   // calling default method
```

```
16.         dm.sayMore("Work is worship");  // calling abstract method
17.
18.         }
19.         }
```

Java forEach 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

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

Java 8 forEach() example 1

```
1.          import java.util.ArrayList;
2.          import java.util.List;
3.          public class ForEachExample {
4.          public static void main(String[] args) {
5.          List<String> gamesList = new ArrayList<String>();
6.          gamesList.add("Football");
7.          gamesList.add("Cricket");
8.          gamesList.add("Chess");
9.          gamesList.add("Hocky");
10.         System.out.println("------------Iterating by passing lambda expression--------------");
11.         gamesList.forEach(games -> System.out.println(games));
12.
13.         }
14.         }
```

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

```
1.  import java.io.FileNotFoundException;
2.  import java.io.FileOutputStream;
3.  public class FinalVariable {
4.  public static void main(String[] args) throws FileNotFoundException {
5.  try(FileOutputStream fileStream=new FileOutputStream("javatpoint.txt");){
6.  String greeting = "Welcome to javaTpoint.";
7.  byte b[] = greeting.getBytes();
8.  fileStream.write(b);
9.  System.out.println("File written");
10. }catch(Exception e) {
11. System.out.println(e);
12. }
13. }
        }
```

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;
```

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

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.

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?

Java 9 Anonymous Inner Classes Improvement

Java 9 introduced a new feature that allows us to use diamond operator with anonymous classes. Using the diamond with anonymous classes was not allowed in Java 7.

In Java 9, as long as the inferred type is denotable, we can use the diamond operator when we create an anonymous inner class.

Data types that can be written in Java program like int, String etc are called denotable types. Java 9 compiler is enough smart and now can infer type.

Note: This feature is included to Java 9, to add type inference in anonymous inner classes.

Let's see an example, in which we are using diamond operator with inner class without specifying type.

Java 9 Anonymous Inner Classes Example

```
1.  abstract class ABCD<T>{
2.  abstract T show(T a, T b);
3.  }
4.  public class TypeInferExample {
5.  public static void main(String[] args) {
6.  ABCD<String> a = new ABCD<>() { // diamond operator is empty, compiler infer type
7.  String show(String a, String b) {
8.  return a+b;
9.  }
10. };
11. String result = a.show("Java","9");
12. System.out.println(result);
13. }
```

14. }

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);
}
}
```

What is Local Variable type inference?

Local variable type inference is a feature in Java 10 that allows the developer to skip the type declaration associated with local variables (those defined inside method definitions, initialization blocks, for-loops, and other blocks like if-else), and the type is inferred by the JDK. It will, then, be the job of the compiler to figure out the datatype of the variable.

```
class A {
public static void main(String a[])
{
var x = "Hi there";
System.out.println(x)
}
}
```

Java Switch Statement

The Java *switch statement* executes one statement from multiple conditions. It is like if-else-if ladder statement. The switch statement works with byte, short, int, long, enum types, String and some wrapper types like Byte, Short, Int, and Long. Since Java 7, you can use strings in the switch statement.

```
1.  public class SwitchExample {
2.  public static void main(String[] args) {
3.  //Declaring a variable for switch expression
4.  int number=20;
5.  //Switch expression
6.  switch(number){
7.  //Case statements
8.  case 10: System.out.println("10");
```

```
9.          break;
10.         case 20: System.out.println("20");
11.         break;
12.         case 30: System.out.println("30");
13.         break;
14.         //Default case statement
15.         default:System.out.println("Not in 10, 20 or 30");
16.         }
17.         }
18.         }
```

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""";
```

RECORD

As developers and software engineers, our aim is to always design ways to obtain maximum efficiency and if we need to write less code for it, then that's a blessing.

In Java, a record is a special type of class declaration aimed at reducing the boilerplate code. Java records were introduced with the intention to be used as a fast way to create data carrier classes, i.e. the classes whose objective is to simply contain data and carry it between modules, also known as POJOs (Plain Old Java Objects) and DTOs (Data Transfer Objects). Record was introduced in Java SE 14 as a preview feature, which is a feature whose design, implementation, and specification are complete but it is not a permanent addition to the language, which means that the feature may or may not exist in the future versions of the language. Java SE 15 extends the preview feature with additional capabilities such as local record classes.

In Java, we have the concept of abstract classes. It is mandatory to inherit from these classes since objects of these classes cannot be instantiated. On the other hand, there is a concept of a final class in Java, which cannot be inherited or extended by any other class. What if we want to restrict the number of classes that can inherit from a particular class? The answer is sealed class. So, a sealed class is a technique that limits the number of classes that can inherit the given class. This means that only the classes designated by the programmer can inherit from that particular class, thereby restricting access to it. when a class is declared sealed, the programmer must specify the list of classes that can inherit it. The concept of sealed classes in Java was introduced in Java 15.

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.

Switch Expressions

Switch expressions were introduced as a preview feature in Java 12 and became a standard feature in Java 14. They provide a more concise and expressive way to use the switch

statement, allowing it to be used as either a statement or an expression. Here's a summary of the key points about switch expressions in Java:

Introduction to Switch Expressions:

Switch expressions in Java allow the switch statement to be used as an expression, resulting in a single value.

They can use either traditional case : labels with fall through or new case ... -> labels without fall through.

Syntax and Usage:

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.

The yield statement can be used to specify the value of a switch expression.

Example:

Here's an example of a switch expression in Java:

```
int numLetters = 0;
Day day = Day.WEDNESDAY;
numLetters = switch (day) {
    case MONDAY, FRIDAY, SUNDAY -> 6;
    case TUESDAY -> 7;
    case THURSDAY, SATURDAY -> 8;
    case WEDNESDAY -> 9;
    default -> throw new IllegalStateException("Invalid day: " + day);
};
System.out.println(numLetters);
```

Benefits:

Switch expressions provide a more concise and readable alternative to traditional switch statements, especially when the goal is to produce a result.

They eliminate the need for break statements and provide a more functional programming style.

Evolution of Switch Expressions:

Switch expressions were introduced as a preview feature in Java 12 and became a standard feature in Java 14.

The feature was designed to simplify everyday coding and prepare the way for the use of pattern matching

in switch statements.

Switch expressions in Java provide a more concise and expressive way to use the switch statement, allowing it to be used as either a statement or an expression. This feature simplifies everyday coding and prepares the way for the use of pattern matching in switch statements.

Yield Keyword

The yield keyword is used in various programming languages, including Python, JavaScript, C#, and Java, with slightly different functionalities in each language. Here's a summary of the yield keyword in different programming languages based on the provided search results:

in Java, the yield keyword is used within switch expressions to exit the switch and return a value that becomes the value of the switch expression. It allows the value of a switch expression to be assigned to a variable and provides an implicit check for covering all cases in the switch expression. The yield keyword is used in different programming languages to create generator functions, pause and resume generator functions, provide values in iterations, and exit switch expressions with a return value. Each language has its own specific use case and syntax for the yield keyword.

Text Blocks

Text blocks in Java are a feature introduced in Java 13 as a preview and became a permanent feature in Java 15. They provide a more concise and readable way to declare multi-line strings in Java. Here's a summary

of the key points about text blocks in Java based on the provided search results:

Purpose of Text Blocks:

The principal purpose of text blocks is to provide clarity by minimizing the Java syntax required to render

a string that spans multiple lines.

Text blocks eliminate the need for explicit line terminators, string concatenations, and delimiters, allowing for the embedding of code snippets and text sequences more or less as-is.

Usage:

Text blocks start with """ followed by optional whitespaces and a newline. For example:

String example = """

Example text""";

Inside the text blocks, newlines and quotes can be used without the need for escaping line breaks. This allows for the inclusion of literal fragments of HTML, JSON, SQL, or other content in a more elegant and readable way.

Benefits:

Text blocks provide a more efficient and readable way to declare multi-line strings in Java, especially when dealing with large or complex text content.

They enhance code readability and maintainability by eliminating the need for explicit escape characters and concatenations.

Evolution of Text Blocks:

Text blocks were introduced as a preview feature in Java 13 and refined in a second preview before becoming a permanent feature in Java 15.

The feature was aimed at reducing the complexity of declaring and using multi-line string literals in Java.

Text blocks in Java provide a more efficient and readable way to declare multi-line strings, enhancing code readability and maintainability. They eliminate the need for explicit escape characters and concatenations, making it easier to include literal fragments of various content types in a more elegant and readable way.

Records

In Java, a record is a new type of class introduced in Java 14 that is designed to be a simple and concise way to declare classes that are transparent holders for shallowly immutable data. Here's a summary of the key points about records in Java:

Purpose of Records:

Records are designed to provide a compact syntax for declaring classes that are meant to be used primarily for storing data.

They are immutable by default, meaning that their state cannot be changed after they are constructed.

Syntax and Usage:

Records are declared using the record keyword followed by the name of the record and a list of components, which are the data fields of the record.

Records automatically provide implementations for methods such as equals(), hashCode(), and toString(), based on the components of the record.

Example:

Here's an example of a record declaration in Java:

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

In this example, the Point record declares two components, x and y, and automatically provides implementations for equals(), hashCode(), and toString().

Benefits:

Records provide a more concise and readable way to declare classes that are intended for data storage, reducing the amount of boilerplate code that needs to be written.

They promote immutability and are well-suited for use in functional programming and concurrent environments.

Evolution of Records:

Records were introduced as a preview feature in Java 14 and became a standard feature in Java 16.

The feature was aimed at simplifying the development of classes meant for data storage and promoting best practices for immutability and transparency.

Records in Java provide a concise and immutable way to declare classes that are primarily used for storing data. They automatically generate implementations for common methods, reducing the need for boilerplate code and promoting best practices for immutability and transparency.

Sealed Classe

In Java, sealed classes were introduced as a preview feature in Java 15 and became a standard

feature in Java 17.

Sealed classes provide a way to control the inheritance hierarchy of a class or interface by specifying

Which classes can extend or implement it.

Here's a summary of the key points about sealed classes in Java based on the provided search results:

1. **Syntax and Usage**:
    - Sealed classes are declared using the `sealed` modifier in their declaration.
    - After any `extends` and `implements` clauses, the `permits` clause specifies the classes that are permitted to extend the sealed class or implement the sealed interface.

2. **Purpose and Benefits**:
    - Sealed classes provide a more fine-grained control over inheritance in Java, allowing classes and interfaces to define their permitted subtypes.
    - This feature is useful for domain modeling and increasing the security of libraries, as it enables the specification of which classes can implement or extend a particular class or interface.
3. **Evolution and Status**:
    - Sealed classes were proposed by JEP 360 and delivered in JDK 15 as a preview feature. They were refined and delivered in JDK 16 as a preview feature with no changes from JDK 16 to JDK 17.
    - With the release of Java 17, sealed classes have become a permanent feature, providing more control over the inheritance hierarchy.

Sealed classes in Java offer a mechanism for controlling class hierarchies, which helps prevent unauthorized extensions and provides a more secure and maintainable codebase.
They enable fine-grained control over which classes can extend or implement a particular class or interface, enhancing the security and flexibility of Java applications.

Important questions

1.How do records simplify the creation of classes for data representation?

2.Explain the concept of module declaration in Java modules.

3.How are repeating annotations used to annotate elements multiple times?

4.How can Base64 encoding and decoding be performed in Java?

5.How is the Stream API used to process collections of objects?