JAVA Features - 5, 7 TO13

JAVA 5 (SOURCE – JAVATPOINT) 5

For-each loop 5

Variable Argument (Varargs) 5

Static Import 5

Autoboxing and Unboxing 6

Enum 6

Covariant Return Type 6

Annotation 7

Generics 7

JAVA 7 (SOURCE - JAVAREVISITED) 8

Type inference 8

Automatic Resource Management (try-with-resources) 8

Underscore in Numeric literals 9

Catching Multiple Exception Type in Single Catch Block 9

Binary Literals with prefix "0b" 10

Java NIO 2.0 10

G1 Garbage Collector 10

Precise Rethrowing of Exception 10

JAVA 8 (SOURCE – JAVATPOINT | JOURNALDEV)........12

Java 8 Date/Time API........12

Lambda Expressions........12

Java Method References........14

Java Functional Interfaces , Static and Default methods........14

Java 8 Stream........14

Java Base64 Encode and Decode........15

forEach() method in Iterable interface........16

Optional Class........17

Collection API improvements........17

Concurrency API........18

Java IO improvements........18

JAVA 9 (SOURCE – JOURNALDEV)........18

Factory Methods for Immutable List, Set, Map and Map.Entry........18

Private methods in Interfaces........19

Java 9 REPL (JShell)........19

Module System........20

Process API Improvements........22

Try With Resources Improvement........22

Optional Class Improvements........23

Stream API Improvements........24

Reactive Streams........25

Enhanced @Deprecated annotation........25

HTTP Client........26

JAVA 10 (SOURCE – HOWTODOINJAVA)........26

JEP 286: Local Variable Type Inference 26

JEP 322: Time-Based Release Versioning 27

JEP 304: Garbage-Collector Interface 27

JEP 307: Parallel Full GC for G1 27

JEP 316: Heap Allocation on Alternative Memory Devices 28

JEP 296: Consolidate the JDK Forest into a Single Repository 28

JEP 314: Additional Unicode Language-Tag Extensions 28

JEP 319: Root Certificates 29

JEP 317: Experimental Java-Based JIT Compiler 29

JEP 312: Thread-Local Handshakes 29

JEP 313: Remove the Native-Header Generation Tool 29

JEP 310: Application Class-Data Sharing 29

JAVA 11 (SOURCE – JOURNALDEV) 29

JEP 330 - Running Java File with single command 29

Local-Variable Syntax for Lambda Parameters 29

Java String Methods 30

Nested Based Access Control 31

Reading/Writing Strings to and from the Files 32

JEP 329: ChaCha20 and Poly1305 Cryptographic Algorithms 32

JEP 315: Improve Aarch64 Intrinsics 32

JEP 333: ZGC: A Scalable Low-Latency Garbage Collector (Experimental) 32

JEP 335: Deprecate the Nashorn JavaScript Engine 32

JEP 309: Dynamic Class-File Constants 32

JEP 318: Epsilon: A No-Op Garbage Collector 32

JEP 320: Remove the Java EE and CORBA Modules 32

JEP 328: Flight Recorder 32

JEP 321: HTTP Client 32

JAVA 12 (SOURCE – JOURNALDEV) 32

Switch Expressions 32

File mismatch() Method 33

Teeing Collectors in Stream API 34

Java Strings New Methods 34

JEP 305: Pattern Matching for instanceof 35

Raw String Literals is Removed From JDK 12. 35

JEP 334: JVM Constants API 35

Compact Number Formatting 35

JVM Changes – JEP 189, JEP 346, JEP 344, and JEP 230. 35

JAVA 13 (SOURCE – JOURNALDEV) 35

Text Blocks – JEP 355 35

New Methods in String Class for Text Blocks 37

Switch Expressions Enhancements – JEP 354 38

FileSystems.newFileSystem() Method 40

DOM and SAX Factories with Namespace Support 40

Reimplement the Legacy Socket API – JEP 353 40

Dynamic CDS Archive – JEP 350 40

ZGC: Uncommit Unused Memory – JEP 351 40

Support for Unicode 12.1 40

Java 5 (Source – javatpoint)

☐ For-each loop

It provides an alternative approach to traverse the array or collection in Java. It is mainly used to traverse the array or collection elements

```
for(int I : arr){
    System.out.println(i);
 }
```

☐ Variable Argument (Varargs)

The varrags allows the method to accept zero or multiple arguments.

Before varargs either we use overloaded method or take an array as the method parameter, but it was not considered good because it leads to the maintenance problem. If we don't know how many argument we will have to pass in the method, varargs is the better approach.

```
  //Method with varargs
     public  static void validates(String... values){
           for(String s:values){
             System.out.println(s);
            }
         }
      }
//Calling varargs method with diff arguments
validates ();//zero argument
validates ("my","name","is","varargs");//four arguments
```

Rules for varargs:

- There can be only one variable argument in the method.
- Variable argument (varargs) must be the last argument.

☐ Static Import

To access any static member of a class directly. There is no need to qualify it by the class name.

```
import static java.lang.System.*;
class StaticImportExample{
  public static void main(String args[]){

   out.println("Hello"); //Now no need of System.out
   out.println("Java");
  }
```

☐ Autoboxing and Unboxing

The automatic conversion of primitive data types into its equivalent Wrapper type is known as boxing and opposite operation is known as unboxing.

```
int a=50;
Integer a2=new Integer(a);//Boxing
Integer a3=5;//Boxing

Integer i=new Integer(50);
int a=i;  ;//UnBoxing
```

☐ Enum

Enum in Java is a data type which contains a fixed set of constants, Enums are used to create our own data type like classes. The enum data type (also known as Enumerated Data Type) is used to define an enum in Java.

```
public enum Browsers {
IE,
Chorme,
FireFox,
Safari
}

//Traversing the enum
for (Browsers  tempBrowser : Browsers .values())  {
System.out.println(tempBrowser );
}
//Specific enum
System.out.println(Browsers.IE);
```

☐ Covariant Return Type

The covariant return type specifies that the return type may vary in the same direction as the subclass.

Before Java5, it was not possible to override any method by changing the return type. But now,

Since Java5, it is possible to override method by changing the return type if subclass overrides any method whose return type is Non-Primitive, but it changes its return type to subclass type. Let's take a simple example:

```
class WebDriver{
          WebDriver get(){
              return this;
             }
}
class ChromeDriver extends WebDriver {
        ChromeDriver  get(){
             return this;
         }
}
```

☐ **Annotation**

Java **Annotation** is a tag that represents the *metadata* i.e. attached with class, interface, methods or fields to indicate some additional information which can be used by java compiler and JVM.
Annotations in Java are used to provide additional information, so it is an alternative option for XML and Java marker interfaces.
There are several built-in annotations in Java. Some annotations are applied to Java code and some to other annotations.

Built-In Java Annotations used in Java code

- @Override
- @SuppressWarnings
- @Deprecated

Built-In Java Annotations used in other annotations

- @Target
- @Retention
- @Inherited
- @Documented

☐ **Generics**

To deal with type-safe objects. It makes the code stable by detecting the bugs at compile time.
Before generics, we can store any type of objects in the collection, i.e., non-generic. Now generics force the java programmer to store a specific type of objects.

1) Type-safety: We can hold only a single type of objects in generics. It doesn?t allow to store other objects, Without Generics, we can store any type of objects.

```
List list = new ArrayList();
list.add(10);
list.add("10");
With Generics, it is required to specify the type of object we need to store.
List<Integer> list = new ArrayList<Integer>();
list.add(10);
list.add("10");// compile-time error
```

2) Type casting is not required: There is no need to typecast the object, Before Generics, we need to type cast.

```
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);//typecasting
```

After Generics, we don't need to typecast the object.

```
List<String> list = new ArrayList<String>();
list.add("hello");
String s = list.get(0);
```

Java 7 (Source - javarevisited)

☐ Type inference

< JAVA 7

```
Map<String, List<String>> employees =  new HashMap<String, List<String>>();
List<Integer> accounts = new ArrayList<Integer>();
```

>JAVA 7

```
Map<String, List<String>> employees =  new HashMap<>();
List<Integer> accounts = new ArrayList<>();
```

☐ String in Switch

< JAVA 7, only integral types can be used as selector for switch-case statement,
> JAVA 7, can use a String object as the selector,

```
String state = "NEW";
switch (state) {
   case "NEW": System.out.println("New"); break;
   case "CANCELED": System.out.println("Cancelled"); break;
   case "REPLACE": System.out.println("Replaced"); break;
   case "FILLED": System.out.println("Filled"); break;
   default: System.out.println("Default");
}
```

☐ Automatic Resource Management (try-with-resources)

Now in Java 7, you can use try-with-resource feature to automatically close resources, which implements AutoClosable and Closeable interface e.g. Streams, Files, Socket handles, database connections etc.
which ensures that each of the resources in try(resources) is closed at the end of the statement by calling close() method of AutoClosable.

```
try (FileInputStream fin = new FileInputStream("info.xml");
  BufferedReader br = new BufferedReader(new InputStreamReader(fin));) {
  if (br.ready()) {
   String line1 = br.readLine();
   System.out.println(line1);
  }
  }catch (FileNotFoundException ex) {
   System.out.println("Info.xml is not found");
  } catch (IOException ex) {
   System.out.println("Can't read the file");
  }
}
```

☐ **Underscore in Numeric literals**

> JAVA 7
you could insert underscore(s) '_' in between the digits in a numeric literal (integral and floating-point literals) to improve readability. This is especially valuable for people who uses large numbers in source files, may be useful in finance and computing domains. For example,

```
✓ int billion = 1_000_000_000;  // 10^9
✓ long creditCardNumber =  1234_4567_8901_2345L; //16 digit number
✓ long ssn = 777_99_8888L;
✓ double pi = 3.1415_9265;
✓ float  pif = 3.14_15_92_65f;
```

By the way remember that you cannot put underscore, just after decimal number or at the beginning or at the end of number. For example, following numeric literals are invalid, because of wrong placement of underscore:

```
× double pi = 3._1415_9265; // underscore just after decimal point
× long creditcardNum = 1234_4567_8901_2345_L; //underscore at the end of
  number
× long ssn = _777_99_8888L; //undersocre at the beginning
```

You can put underscore at convenient points to make it more readable,
- ➢ Large amounts putting underscore between three digits make sense,
- ➢ Credit card numbers, which are 16 digit long, putting underscore after 4th digit make sense, as they are printed in cards.

☐ **Catching Multiple Exception Type in Single Catch Block**

> JAVA 7, A single catch block can handle more than one exception types

```
try {
     ……
    } catch(ClassNotFoundException|SQLException ex) {
   ex.printStackTrace();
      }
```

By the way, just remember that Alternatives in a multi-catch statement cannot be related by sub classing. For example, a multi-catch statement like below will throw compile time error :

```
catch (FileNotFoundException | IOException ex) {
   ex.printStackTrace();
}
```

Alternatives in a multi-catch statement cannot be related by sub classing, it will throw error at compile time : java.io.FileNotFoundException is a subclass of alternative java.io.IOException
    at Test.main(Test.java:18)

- ## Binary Literals with prefix "0b"

> JDK 7, you can express literal values in binary with prefix '0b' (or '0B') for integral types (byte, short, int and long), similar to C/C++ language. Before JDK 7, you can only use octal values (with prefix '0') or hexadecimal values (with prefix '0x' or '0X').

```
int mask = 0b01010000101;
```

or even better

```
int binary = 0B0101_0000_1010_0010_1101_0000_1010_0010;
```

- ## Java NIO 2.0

Java SE 7 introduced java.nio.file package and its related package, java.nio.file.attribute, provide comprehensive support for file I/O and for accessing the default file system. It also introduced the Path class which allow you to represent any path in operating system.

New File system API complements older one and provides several useful method checking, deleting, copying, and moving files. for example, now you can check if a file is hidden in Java. You can also create symbolic and hard links from Java code. JDK 7 new file API is also capable of searching for files using wild cards. You also get support to watch a directory for changes. I would recommend to check Java doc of new file package to learn more about this interesting useful feature.

- ## G1 Garbage Collector

JDK 7 introduced a new Garbage Collector known as G1 Garbage Collection, which is short form of garbage first. G1 garbage collector performs clean-up where there is most garbage. To achieve this it split Java heap memory into multiple regions as opposed to 3 regions in the prior to Java 7 version (new, old and permgen space). It's said that G1 is quite predictable and provides greater through put for memory intensive applications.

- ## Precise Rethrowing of Exception

The Java SE 7 compiler performs more precise analysis of re-thrown exceptions than earlier releases of Java SE. This enables you to specify more specific exception types in the throw's clause of a method declaration. before JDK 7, re-throwing an exception was treated as throwing the type of the catch parameter. For example, if your try block can throw ParseException as well as IOException. In order to catch all exceptions and rethrow them, you would have to catch Exception and declare your method as throwing an Exception. This is sort of obscure non-precise throw, because you are throwing a general Exception type (instead of specific ones) and statements calling your method need to catch this general Exception. This will be clearer by seeing following example of exception handling in code prior to Java 1.7

```
public void obscure() throws Exception{
    try {
        new FileInputStream("abc.txt").read();
        new SimpleDateFormat("ddMMyyyy").parse("12-03-2014");
    } catch (Exception ex) {
        System.out.println("Caught exception: " + ex.getMessage());
        throw ex;
    }
}
```

From JDK 7 onwards you can be more precise while declaring type of Exception in throws clause of any method. This precision in determining which Exception is thrown from the fact that, If you re-throw an exception from a catch block, you are actually throwing an exception type which:

1) your try block can throw,
2) has not handled by any previous catch block, and
3) is a subtype of one of the Exception declared as catch parameter

This leads to improved checking for re-thrown exceptions. You can be more precise about the exceptions being thrown from the method and you can handle them a lot better at client side, as shown in following example :

```
public void precise() throws ParseException, IOException {
    try {
        new FileInputStream("abc.txt").read();
        new SimpleDateFormat("ddMMyyyy").parse("12-03-2014");
    } catch (Exception ex) {
        System.out.println("Caught exception: " + ex.getMessage());
        throw ex;
    }
}
```

The Java SE 7 compiler allows you to specify the exception types ParseException and IOException in the throws clause in the preciese() method declaration because you can re-throw an exception that is a super-type of any of the types declared in the throws, we are throwing java.lang.Exception, which is super class of all checked Exception. Also in some places you will see final keyword with catch parameter, but that is not mandatory any more.

That's all about what you can revise in JDK 7. All these new features of Java 7 are very helpful in your goal towards clean code and developer productivity. With lambda expression introduced in Java 8, this goal to cleaner code in Java has reached another milestone. Let me know, if you think I have left out any useful feature of Java 1.7, which you think should be here.

JAVA 8 (Source – javatpoint | journaldev)

- ☐ **Java 8 Date/Time API**

Java has introduced a new Date and Time API since Java 8. The java.time package contains Java 8 Date and Time classes.

- java.time.LocalDate class
- java.time.LocalTime class
- java.time.LocalDateTime class
- java.time.MonthDay class
- java.time.OffsetTime class
- java.time.OffsetDateTime class
- java.time.Clock class
- java.time.ZonedDateTime class
- java.time.ZoneId class
- java.time.ZoneOffset class
- java.time.Year class
- java.time.YearMonth class
- java.time.Period class
- java.time.Duration class
- java.time.Instant class
- java.time.DayOfWeek enum
- java.time.Month enum

- ☐ **Lambda Expressions**

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.

(argument-list) -> {body}

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.

Without Lambda Expression

```
interface Drawable{
    public void draw();
}
public class LambdaExpressionExample {
    public static void main(String[] args) {
        int width=10;

        //without lambda, Drawable implementation using anonymous class
        Drawable d=new Drawable(){
            public void draw(){System.out.println("Drawing "+width);}
        };
        d.draw();
    }
}
```

Output:

```
Drawing 10
```

Java Lambda Expression Example

Now, we are going to implement the above example with the help of Java lambda expression.

```
@FunctionalInterface  //It is optional
interface Drawable{
    public void draw();
}

public class LambdaExpressionExample2 {
    public static void main(String[] args) {
        int width=10;

        //with lambda
        Drawable d2=()->{
            System.out.println("Drawing "+width);
        };
        d2.draw();
    }
}
```

Output:

```
Drawing 10
```

☐ 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. In this tutorial, we are explaining method reference concept in detail.

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 Functional Interfaces , Static and Default methods

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.

```
interface comparable{
     void say(String msg);   // abstract method
    default void compare(){
        System.out.println("Default compare");
    }
    Static void gets(String msg){
        System.out.println("Static ");
    }
}
```

☐ 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 provides following features:

- 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 applied various operations with the help of stream.

```
List<Float> productPriceList2 =productsList.stream()
                              .filter(p -> p.price > 30000)// filtering data
                              .map(p->p.price)       // fetching price
                              .collect(Collectors.toList()); // collecting as list
```

- ☐ **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.

```
import java.util.Base64;
```

```
public class Base64BasicEncryptionExample {
    public static void main(String[] args) {
        // Getting encoder
        Base64.Encoder encoder = Base64.getEncoder();
        // Creating byte array
        bytebyteArr[] = {1,2};
        // encoding byte array
        bytebyteArr2[] = encoder.encode(byteArr);
        System.out.println("Encoded byte array: "+byteArr2);
        bytebyteArr3[] = newbyte[5];// Make sure it has enough size to store copied bytes

        intx = encoder.encode(byteArr,byteArr3);    // Returns number of bytes written
        System.out.println("Encoded byte array written to another array: "+byteArr3);
        System.out.println("Number of bytes written: "+x);

        // Encoding string
        String str = encoder.encodeToString("JavaTpoint".getBytes());
        System.out.println("Encoded string: "+str);
        // Getting decoder
        Base64.Decoder decoder = Base64.getDecoder();
        // Decoding string
        String dStr = new String(decoder.decode(str));
        System.out.println("Decoded string: "+dStr);
    }
}
```

- **forEach() method in Iterable interface**

Java 8 has introduced *forEach* method in java.lang.Iterable interface so that while writing code we focus on business logic only. *forEach* method takes java.util.function.Consumer object as argument, so it helps in having our business logic at a separate location that we can reuse. Let's see forEach usage with simple example.

```
//traversing using Iterator
Iterator<Integer> it = myList.iterator();
while(it.hasNext()){
        Integer i = it.next();
        System.out.println("Iterator Value::"+i);
}
//traversing through forEach method of Iterable with anonymous class
myList.forEach(new Consumer<Integer>() {
```

```
            public void accept(Integer t) {
                System.out.println("forEach anonymous class Value::"+t);
            }
    });

    //Lambada used for forEach – 1st Way
    myList.forEach((Integer t ) -> System.out.println(t); });

    //Lambada used for forEach – 2nd  Way
    myList.forEach( t -> System.out.println(t) );
```

☐
Optional Class

It is a public final class and used to deal with NullPointerException in Java application. You must import java.util package to use this class. It provides methods which are used to check the presence of value for particular variable.

```
import java.util.Optional;
public class OptionalExample {
    public static void main(String[] args) {
        String[] str = new String[10];
        Optional<String> checkNull = Optional.ofNullable(str[5]);
        if(checkNull.isPresent()){  // check for value is present or not
            String lowercaseString = str[5].toLowerCase();
            System.out.print(lowercaseString);
        }else
            System.out.println("string value is not present");
    }
}
```

☐
Collection API improvements

We have already seen forEach() method and Stream API for collections. Some new methods added in Collection API are:

Iterator default method forEachRemaining(Consumer action) to perform the given action for each remaining element until all elements have been processed or the action throws an exception.
Collection default method removeIf(Predicate filter) to remove all of the elements of this collection that satisfy the given predicate.
Collection spliterator() method returning Spliterator instance that can be used to traverse elements sequentially or parallel.
Map replaceAll(), compute(), merge() methods.

Performance Improvement for HashMap class with Key Collisions

- ## Concurrency API

Some important concurrent API enhancements are:

ConcurrentHashMap compute(), forEach(), forEachEntry(), forEachKey(), forEachValue(), merge(), reduce() and search() methods.
CompletableFuture that may be explicitly completed (setting its value and status).
Executors newWorkStealingPool() method to create a work-stealing thread pool using all available processors as its target parallelism level.

- ## Java IO improvements

Some IO improvements known to me are:

Files.list(Path dir) that returns a lazily populated Stream, the elements of which are the entries in the directory.
Files.lines(Path path) that reads all lines from a file as a Stream.
Files.find() that returns a Stream that is lazily populated with Path by searching for files in a file tree rooted at a given starting file.
BufferedReader.lines() that return a Stream, the elements of which are lines read from this BufferedReader.

JAVA 9 (Source – journaldev)

- ## Factory Methods for Immutable List, Set, Map and Map.Entry

In Java 9, introduced some convenient factory methods to create Immutable List, Set, Map and Map.Entry objects. These utility methods are used to create empty or non-empty Collection objects.

List and Set interfaces have "of()" methods to create an empty or no-empty Immutable List or Set objects as shown below:

LIST

< Java 9

```
List<String> emptyList = new ArrayList<>();
 List<String> immutableList =
Collections.unmodifiableList(emptyList);
```

> Java 9

```
jshell> List<String> immutableList = List.of();
jshell> List immutableList = List.of("one","two","three");
```

SET

```
jshell> Set<String> immutableSet = Set.of();
jshell> Set<String> immutableSet = Set.of("one","two","three");
```

MAP

```
jshell> Map<Integer,String> emptyImmutableMap = Map.of()
emptyImmutableMap ==> {}
jshell> Map<Integer,String> nonemptyImmutableMap = Map.of(1,
"one", 2, "two", 3, "three")
```

- ☐

Private methods in Interfaces

Private methods and private static methods are added in Java 9 interface changes.\

Java 9 private methods in interfaces have following benefits,

- ➢ No need to write duplicate code, hence more code reusability.
- ➢ We got the choice to expose only our intended methods implementations to clients.

```
public interface Card{

  private Long createCardID(){
    // Method implementation goes here.
  }

  private static void displayCardDetails(){
    // Method implementation goes here.
  }

}
```

- ☐

Java 9 REPL (JShell)

introduced a new tool called "jshell". It stands for Java Shell and known as REPL (Read Evaluate Print Loop). Many languages already feature an interactive Read-Eval-Print-Loop, and Java now joins this club. It is used to execute and test any Java Constructs like class, interface, enum, object, statements etc. very easily. You can launch jshell from the console and directly start typing and executing Java code. The immediate feedback of jshell makes it a great tool to explore APIs and try out language features.

```
jshell> System.out.println("Hello World");

Hello World

jshell> String str = "Hello JournalDev Users"

str ==> "Hello JournalDev Users"
```

```
jshell> str
str ==> "Hello JournalDev Users"
jshell> 10/5
$2 ==> 2
jshell> 10.0/3
$4 ==> 3.3333333333333335
```

- ## Module System

Jigsaw project is going to introduce completely new concept of Java SE 9: **Java Module System**.

It is very big and prestigious project from Oracle Corp in Java SE 9 release. Initially, they have started this project as part of Java SE 7 Release. However, with huge changes, it's postponed to Java SE 8 then again postponed. Now it is about to release with Java SE 9 in September 2017.

Main Goals of Jigsaw Project:

The Modular JDK

As we know, Current JDK system is too big. So they have decided to divide JDK itself into small modules to get a number of benefits (We will discuss them soon in the coming sections).

Modular Source Code

Current source code jar files are too big, especially rt.jar is too big right. So they are going to divide Java Source code into smaller modules.

Modular Run-Time Images

The main goal of this Feature is "Restructure the JDK and JRE run-time images to accommodate modules".

Encapsulate Most Internal APIs

The main goal of this feature is "Make most of the JDK's internal APIs inaccessible by default but leave a few critical, widely-used internal APIs accessible".

Java Platform Module System

The main goal of this Feature is "Allowing the user to create their modules to develop their applications".

jlink: The Java Linker

The main goal of this jlink Tool is “Allowing the user to create executable to their applications”.

Advantages of Java SE 9 Module System

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

As Java SE 9 is going to divide JDK, JRE, JARs etc, into smaller modules, we can use whatever modules we want. So it is very easy to scale down the Java Application to Small devices.

- Ease of Testing and Maintainability.
- Supports better Performance.
- As public is not just public, it supports very Strong Encapsulation. (Don’t worry its a big concept. we will explore it with some useful examples soon).
- We cannot access Internal Non-Critical APIs anymore.
- Modules can hide unwanted and internal details very safely, we can get better Security.
- Application is too small because we can use only what ever modules we want.
- Its easy to support Less Coupling between components.
- Its easy to support Single Responsibility Principle (SRP).

HelloWorld.java

```
package com.hello;
public class HelloWorld {
  public String sayHelloWorld() {
      return "Hello World!";
  }

}
```

Develop Module Descriptor at Module root folder “com.hello”.

module-info.java

```
module com.hello {
  exports com.hello;
 }
```

HelloWorldClient.java

```
package com.hello.client;
import com.hello.HelloWorld;
public class HelloWorldClient {
  public static void main (String arg[]) {
    HelloWorld hello = new HelloWorld();
    System.out.println(hello.sayHelloWorld());
        }
}
```

Develop Module Descriptor at Module root folder “com.hello”.

module-info.java

```
module com.hello.client {
  requires com.hello;
  }
```

If we observe above Module Descriptor, we can say that “com.hello.client” module is using types which are available in “com.hello” package name. It is not exporting anything to the outside world. So other modules cannot access types defined under “com.hello.client” package name.

☐ Process API Improvements

Java SE 9 is coming with some improvements in Process API. They have added couple new classes and methods to ease the controlling and managing of OS processes.

Two new interfcase in Process API:

java.lang.ProcessHandle
java.lang.ProcessHandle.Info

Process API example

```
 ProcessHandle currentProcess = ProcessHandle.current();
 System.out.println("Current Process Id: = " +
currentProcess.getPid());
```

☐ Try With Resources Improvement

We know, Java SE 7 has introduced a new exception handling construct: Try-With-Resources to manage resources automatically. The main goal of this new statement is “Automatic Better Resource Management”.

Java SE 9 is going to provide some improvements to this statement to avoid some more verbosity and improve some Readability.

Java SE 7 example

```
void testARM_Before_Java9() throws IOException{
 BufferedReader reader1 = new BufferedReader(new
FileReader("journaldev.txt"));
 try (BufferedReader reader2 = reader1) {
   System.out.println(reader2.readLine());
 }
}
```

Java 9 example

```
void testARM_Java9() throws IOException{
 BufferedReader reader1 = new BufferedReader(new
FileReader("journaldev.txt"));
 try (reader1) {
   System.out.println(reader1.readLine());
 }
}
```

☐ **Optional Class Improvements**

In Java SE 9, Oracle Corp has introduced the following three methods to improve Optional functionality.

- stream()
- ifPresentOrElse()
- or()

Java SE 9: Optional stream() Method

If a value present in the given Optional object, this stream() method returns a sequential Stream with that value. Otherwise, it returns an Empty Stream.

They have added "stream()" method to work on Optional objects lazily as shown below:

```
Stream<Optional> emp = getEmployee(id)
Stream empStream = emp.flatMap(Optional::stream)
```

Here Optional.stream() method is used convert a Stream of Optional of Employee object into a Stream of Employee so that we can work on this result lazily in the result code.

Java SE 8 Style: Optional Methods

In Java SE 8, we should use ifPresent(), isPresent(), orElse() etc. methods to check an Optional object and perform some functionality on it. It's bit tedious process to perform this. However, Java SE 9 has introduced a new method to overcome this problem.
Let us explore Java SE 8 style in this section. We will explore that new method in next section.
Here we are going to explore the following three Optional class methods:

ifPresent()

```
void ifPresent(Consumer action)
```

If a value is present, performs the given action with the value, otherwise does nothing.

isPresent()

```
boolean isPresent()
```

If a value is present, returns true, otherwise false.

orElse()

```
public T orElse(T other)
```

If a value is present, returns the value, otherwise returns other.

Example:-

```
import java.util.Optional;

public class JavaSE8OptionalDemo
{
  public static void main(String[] args)
  {
    Optional<Integer> opt1 = division(4,2);
    opt1.ifPresent( x -> System.out.println("Option1: Result 
found = " + x));
    Optional<Integer> opt2 = division(4,0);
    opt2.ifPresent( x -> System.out.println("Option2: Result 
found: " + x));
    System.out.println("Option2: Result not found, default vlaue 
= " + opt2.orElse(new Integer(0)));
    if(opt2.isPresent())
      System.out.println("Option2: Result found.");
    else
      System.out.println("Option2: Result not found.");
  }

  public static Optional<Integer> division(Integer i1,Integer 
i2)
  {
    if(i2 == 0) return Optional.empty();
    else {
      Integer i3 = i1/i2;
      return Optional.of(i3);
    }
  }
}
```

Outuput:-

```
Option1: Result found = 2
Option2: Result not found, default vlaue = 0
Option2: Result not found.
```

☐ **Stream API Improvements**

In Java SE 9, Oracle Corp has added the followng four useful new methods to java.util.Stream interface.

- dropWhile
- takeWhile
- iterate
- ofNullable

As Stream is an interface, first two new methods are default methods and last two are static methods. Two of them are very important: dropWhile and takeWhile methods.

If you are familiar with Scala Language or any Functions programming language, you will definitely know about these methods. These are very useful methods in writing some functional style code.

☐ Reactive Streams

Now-a-days, Reactive Programming has become very popular in developing applications to get some beautiful benefits. Scala, Play, Akka etc. Frameworks has already integrated Reactive Streams and getting many benefits. Oracle Corps is also introducing new Reactive Streams API in Java SE 9.

Java SE 9 Reactive Streams API is a Publish/Subscribe Framework to implement Asynchronous, Scalable and Parallel applications very easily using Java language.

Java SE 9 has introduced the following API to develop Reactive Streams in Java-based applications.

java.util.concurrent.Flow
java.util.concurrent.Flow.Publisher
java.util.concurrent.Flow.Subscriber
java.util.concurrent.Flow.Processor

☐ Enhanced @Deprecated annotation

In Java SE 9, Oracle Corp has enhanced @Deprecated annotation to provide more information about deprecated API and also provide a **Tool** to analyse an application's static usage of deprecated APIs. They have add two methods to this Deprecated interface: **forRemoval** and **since** to serve this information.

- **@Deprecated(since="<*version*>")**
  - <*version*> is the version when the API was deprecated. This is for informational purposes. The default is the empty string ("").
- **@Deprecated(forRemoval=<*boolean*>)**
  - forRemoval=true indicates that the API is subject to removal in a future release.

- forRemoval=false recommends that code should no longer use this API; however, there is no current intent to remove the API. This is the default value.

For example: @Deprecated(since="9", forRemoval=true)

☐ HTTP Client

In Java SE 9, Oracle Corp is going to release New HTTP 2 Client API to support HTTP/2 protocol and WebSocket features. As existing or Legacy HTTP Client API has numerous issues (like supports HTTP/1.1 protocol and does not support HTTP/2 protocol and WebSocket, works only in Blocking mode and lot of performance issues.), they are replacing this HttpURLConnection API with new HTTP client.

They are going to introduce new HTTP 2 Client API under "java.net.http" package. It supports both HTTP/1.1 and HTTP/2 protocols. It supports both Synchronous (Blocking Mode) and Asynchronous Modes. It supports Asynchronous Mode using WebSocket API.

We can see this new API at: http://download.java.net/java/jdk9/docs/api/java/net/http/package-summary.html

HTTP 2 Client Example

```
jshell> import java.net.http.*
jshell> import static java.net.http.HttpRequest.*
jshell> import static java.net.http.HttpResponse.*
jshell> URI uri = new
URI("http://rams4java.blogspot.co.uk/2016/05/java-news.html")
uri ==> http://rams4java.blogspot.co.uk/2016/05/java-news.html
jshell> HttpResponse response =
HttpRequest.create(uri).body(noBody()).GET().response()
response ==> java.net.http.HttpResponseImpl@79efed2d
jshell> System.out.println("Response was " +
response.body(asString()))
```

JAVA 10 (source – howtodoinjava)

☐ JEP 286: Local Variable Type Inference

Java has now `var` style declarations. It allows you to declare a local variable without specifying its type. The type of variable will be inferred from type of actual object created. It claims to be the only real feature for developers in JDK 10. e.g.

```
var str = "Hello world";
```

```
//or
String str = "Hello world";

var i;  //Invalid Declaration - - Cannot use 'var' on variable without
initializer

var j = 10; //Valid Declaration
```

Usage NOT allowed as :

1. Method parameters
2. Constructor parameters
3. Method return types
4. Class fields
5. Catch formals (or any other kind of variable declaration)

- ☐ **JEP 322: Time-Based Release Versioning**

The new format of the version number is:

```
$FEATURE.$INTERIM.$UPDATE.$PATCH
```

If you run command `java -version` in comaand prompt/terminal the you will get outout version information like this:

```
C:\Users\Lokesh>java -version
java version "10.0.1" 2018-04-17
Java(TM) SE Runtime Environment 18.3 (build 10.0.1+10)
Java HotSpot(TM) 64-Bit Server VM 18.3 (build 10.0.1+10, mixed mode)
```

- ☐ **JEP 304: Garbage-Collector Interface**

In earlier JDK structure, the components that made up a Garbage Collector (GC) implementation were scattered throughout various parts of the code base. It's changed in Java 10. Now, it is a clean interface within the JVM source code to allow alternative collectors to be quickly and easily integrated. It will improve source-code isolation of different garbage collectors.

This is purely refactoring. Everything that worked before needs to work afterwards, and performance should not regress.

- ☐ **JEP 307: Parallel Full GC for G1**

Java 9 introduced G1 (garbage first) garbage collector. The G1 garbage collector is designed to avoid full collections, but when the concurrent collections can't reclaim memory fast enough. With this change, a fall back full GC will occur.

The current implementation of the full GC for G1 uses a single threaded mark-sweep-compact algorithm. This change will parallelize the mark-sweep-compact algorithm and use the same number of threads. It will be triggered when concurrent threads for collection can't revive the memory fast enough.

The number of threads can be controlled by the `-XX:ParallelGCThreads` option.

☐ **JEP 316: Heap Allocation on Alternative Memory Devices**

The goal of this change is to enable the HotSpot VM to allocate the Java object heap on an alternative memory device, such as an NV-DIMM, specified by the user.

To allocate the heap in such memory we can add a new option, `-XX:AllocateHeapAt=<path>`. This option would take a path to the file system and use memory mapping to achieve the desired result of allocating the object heap on the memory device. The existing heap related flags such as `-Xmx`, `-Xms`, etc., and garbage-collection related flags would continue to work as before.

☐ **JEP 296: Consolidate the JDK Forest into a Single Repository**

As part of this change numerous repositories of the JDK forest is combined into a single repository in order to simplify and streamline development.

In JDK 9 there are eight repos: `root`, `corba`, `hotspot`, `jaxp`, `jaxws`, `jdk`, `langtools`, and `nashorn`. In the consolidated forests, code for Java modules is generally combined under a single top-level `src` directory. For example, today in the JDK forest there are module-based directories like

```
$ROOT/jdk/src/java.base
...
$ROOT/langtools/src/java.compiler
...
```

In the consolidated forest, this code is instead organized as-

```
$ROOT/src/java.base
$ROOT/src/java.compiler
...
```

☐ **JEP 314: Additional Unicode Language-Tag Extensions**

It's goal is to enhance `java.util.Locale` and related APIs to implement additional Unicode extensions of BCP 47 language tags. Support for BCP 47 language tags was was initially added in Java SE 7, with support for the Unicode locale extension limited to calendars and numbers. This JEP will implement more of the extensions specified in the latest LDML specification, in the relevant JDK classes.

This JEP will add support for the following additional extensions:

- cu (currency type)
- fw (first day of week)
- rg (region override)
- tz (time zone)

☐ **JEP 319: Root Certificates**

☐ **JEP 317: Experimental Java-Based JIT Compiler**

☐ **JEP 312: Thread-Local Handshakes**

☐ **JEP 313: Remove the Native-Header Generation Tool**

☐ **JEP 310: Application Class-Data Sharing**

JAVA 11 (Source – journaldev)

☐ **JEP 330 - Running Java File with single command**

One major change is that you don't need to compile the java source file with javac tool first. You can directly run the file with java command and it implicitly compiles.

☐ **Local-Variable Syntax for Lambda Parameters**

JEP 323, Local-Variable Syntax for Lambda Parameters is the only language feature release in Java 11.
In Java 10, Local Variable Type Inference was introduced. Thus we could infer the type of the variable from the RHS – `var list = new ArrayList<String>();`
JEP 323 allows `var` to be used to declare the formal parameters of an implicitly typed lambda expression.

We can now define :

```
(var s1, var s2) -> s1 + s2
```

This was possible in Java 8 too but got removed in Java 10. Now it's back in Java 11 to keep things uniform.

But why is this needed when we can just skip the type in the lambda?
If you need to apply an annotation just as @Nullable, you cannot do that without defining the type.

Limitation of this feature – You must specify the type var on all parameters or none. Things like the following are not possible:

```
(var s1, s2) -> s1 + s2 //no skipping allowed
(var s1, String y) -> s1 + y //no mixing allowed
var s1 -> s1 //not allowed. Need parentheses if you use var in
lambda.
```

☐ **Java String Methods**

isBlank() – This instance method returns a boolean value. Empty Strings and Strings with only white spaces are treated as blank.

lines()

This method returns a string array which is a collection of all substrings split by lines.

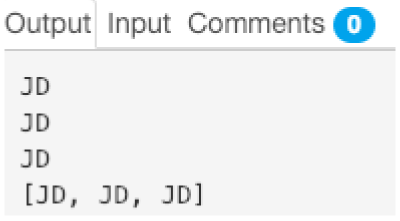
strip(), stripLeading(), stripTrailing()

strip() – Removes the white space from both, beginning and the end of string.

repeat(int)

The repeat method simply repeats the string that many numbers of times as mentioned in the method in the form of an int.

```
import java.util.*;
public class Main {
    public static void main(String[] args) throws Exception {
// isBlank
        System.out.println(" ".isBlank()); //true
        String s = "Anupam";
        System.out.println(s.isBlank()); //false
        String s1 = "";
        System.out.println(s1.isBlank()); //true
// lines
        String str = "JD\nJD\nJD";
        System.out.println(str);
System.out.println(str.lines().collect(Collectors.toList()));
```


Output Input Comments 0

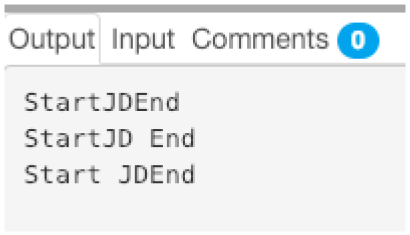
```
JD
JD
JD
[JD, JD, JD]
```

// strip(), stripLeading(), stripTrailing()

```
        String str = " JD ";
        System.out.print("Start");
        System.out.print(str.strip());
        System.out.println("End");

        System.out.print("Start");
        System.out.print(str.stripLeading());
        System.out.println("End");

        System.out.print("Start");
        System.out.print(str.stripTrailing());
        System.out.println("End");
```



```
StartJDEnd
StartJD End
Start JDEnd
```

```
        String str = "=".repeat(2);
        System.out.println(str); //prints ==
}
```

☐ **Nested Based Access Control**

Before Java 11 this was possible:

```
public class Main {
    public void myPublic() {
    }
    private void myPrivate() {
    }
    class Nested {
        public void nestedPublic() {
            myPrivate();
        }
    }
}
```

private method of the main class is accessible from the above-nested class in the above manner.

But if we use Java Reflection, it will give an IllegalStateException.

```
Method method = ob.getClass().getDeclaredMethod("myPrivate");
method.invoke(ob);
```

Java 11 nested access control addresses this concern in reflection.
java.lang.Class introduces three methods in the reflection
API: getNestHost(), getNestMembers(), and isNestmateOf().

- **Reading/Writing Strings to and from the Files**

  Java 11 strives to make reading and writing of String convenient.
  It has introduced the following methods for reading and writing to/from the files:

  readString()
  writeString()

  Following code showcases an example of this

```
Path path = Files.writeString(Files.createTempFile("test",
".txt"), "This was posted on JD");
System.out.println(path);
String s = Files.readString(path);
System.out.println(s); //This was posted on JD
```

- **JEP 329: ChaCha20 and Poly1305 Cryptographic Algorithms**
- **JEP 315: Improve Aarch64 Intrinsics**
- **JEP 333: ZGC: A Scalable Low-Latency Garbage Collector (Experimental)**
- **JEP 335: Deprecate the Nashorn JavaScript Engine**
- **JEP 309: Dynamic Class-File Constants**
- **JEP 318: Epsilon: A No-Op Garbage Collector**
- **JEP 320: Remove the Java EE and CORBA Modules**
- **JEP 328: Flight Recorder**
- **JEP 321: HTTP Client**

JAVA 12 (Source – journaldev)

- **Switch Expressions**

  Java 12 has enhanced Switch expressions for Pattern matching. Introduced in JEP 325, as a preview language feature, the new Syntax is L ->.

Following are some things to note about Switch Expressions:

- The new Syntax removes the need for break statement to prevent fallthroughs.
- Switch Expressions don't fall through anymore.
- Furthermore, we can define multiple constants in the same label.
- default case is now compulsory in Switch Expressions.
- break is used in Switch Expressions to return values from a case itself.
- With the new Switch expression, we don't need to set break everywhere thus prevent logic errors!

With the new Switch expression, we don't need to set break everywhere thus prevent logic errors!

```
String result = switch (day) {
            case "M", "W", "F" -> "MWF";
            case "T", "TH", "S" -> "TTS";
            default -> {
                if(day.isEmpty())
                    break "Please insert a valid day.";
                else
                    break "Looks like a Sunday.";
            }
        };
        System.out.println(result);
```

File mismatch() Method

Java 12 added the following method to compare two files:

```
public static long mismatch(Path path, Path path2) throws IOException
```

This method returns the position of the first mismatch or -1L if there is no mismatch.

Two files can have a mismatch in the following scenarios:

- If the bytes are not identical. In this case, the position of the first mismatching byte is returned.
- File sizes are not identical. In this case, the size of the smaller file is returned.

Example code snippet from IntelliJ Idea is given below:

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;

public class FileMismatchExample {

    public static void main(String[] args) throws IOException {
        Path filePath1 = Files.createTempFile("file1", ".txt");
        Path filePath2 = Files.createTempFile("file2", ".txt");
        Files.writeString(filePath1,"JournalDev Test String");
        Files.writeString(filePath2,"JournalDev Test String");

        long mismatch = Files.mismatch(filePath1, filePath2);

        System.out.println("File Mismatch position... It returns -1 if there is no mismatch");

        System.out.println("Mismatch position in file1 and file2 is >>>>");
        System.out.println(mismatch);

        filePath1.toFile().deleteOnExit();
        filePath2.toFile().deleteOnExit();

        System.out.println();

        Path filePath3 = Files.createTempFile("file3", ".txt");
        Path filePath4 = Files.createTempFile("file4", ".txt");
        Files.writeString(filePath3,"JournalDev Test String");
        Files.writeString(filePath4,"JournalDev.com Test String");

        long mismatch2 = Files.mismatch(filePath3, filePath4);

        System.out.println("Mismatch position in file3 and file4 is >>>>");
        System.out.println(mismatch2);

        filePath3.toFile().deleteOnExit();
        filePath4.toFile().deleteOnExit();
    }
}
```

☐ **Teeing Collectors in Stream API**

Teeing Collector is the new collector utility introduced in the Streams API.

This collector has three arguments – Two collectors and a Bi-function.
All input values are passed to each collector and the result is available in the Bi-function.

```
double mean = Stream.of(1, 2, 3, 4, 5)
                  .collect(Collectors.teeing(
                          summingDouble(i -> i),
                          counting(),
                          (sum, n) -> sum / n));

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

The output is **3.0**.

☐ **Java Strings New Methods**

4 new methods have been introduced in Java 12 which are:

- indent(int n)
- transform(Function f)
- Optional describeConstable()
- String resolveConstantDesc(MethodHandles.Lookup lookup)

☐ **JEP 305: Pattern Matching for instanceof**

Another Preview Language feature!

The old way to typecast a type to another type is:

```
if (obj instanceof String) {
   String s = (String) obj;
   // use s in your code from here
}
```

The new way is :

```
if (obj instanceof String s) {
   // can use s directly here
}
```

This saves us some typecasting which were unnecessary.

☐ **Raw String Literals is Removed From JDK 12.**

☐ **JEP 334: JVM Constants API**

☐ **Compact Number Formatting**

☐ **JVM Changes – JEP 189, JEP 346, JEP 344, and JEP 230.**

JAVA 13 (Source – journaldev)

☐ **Text Blocks – JEP 355**

This is a preview feature. It allows us to create multiline strings easily. The multiline string has to be written inside a pair of triple-double quotes.

The string object created using text blocks has no additional properties. It's an easier way to create multiline strings. We can't use text blocks to create a single-line string.

The opening triple-double quotes must be followed by a line terminator.

```
package com.journaldev.java13.examples;
```

```
public class TextBlockString {
	/**
	 * JEP 355: Preview Feature
	 */
	@SuppressWarnings("preview")
	public static void main(String[] args) {
		String textBlock = """
				Hi
				Hello
				Yes""";

		String str = "Hi\nHello\nYes";

		System.out.println("Text Block String:\n" + textBlock);
		System.out.println("Normal String Literal:\n" + str);

		System.out.println("Text Block and String Literal equals() Comparison: " +
(textBlock.equals(str)));
		System.out.println("Text Block and String Literal == Comparison: " +
(textBlock == str));
	}

}
```

Output:

```
Text Block String:
Hi
Hello
Yes
Normal String Literal:
Hi
Hello
Yes
Text Block and String Literal equals() Comparison: true
Text Block and String Literal == Comparison: true
```

It's useful in easily creating HTML and JSON strings in our Java program.

```
String textBlockHTML = """
		<html>
		<head>
			<link href='/css/style.css' rel='stylesheet' />
		</head>
		<body>
		<h1>Hello World</h1>
	</body>
	</html>""";
```

```
String textBlockJSON = """
                {
                        "name":"Pankaj",
                        "website":"JournalDev"
                }""";
```

☐ **New Methods in String Class for Text Blocks**

There are three new methods in the String class, associated with the text blocks feature.

1. formatted(Object… args): it's similar to the String format() method. It's added to support formatting with the text blocks.
2. stripIndent(): used to remove the incidental white space characters from the beginning and end of every line in the text block. This method is used by the text blocks and it preserves the relative indentation of the content.
3. translateEscapes(): returns a string whose value is this string, with escape sequences translated as if in a string literal.
4. equences translated as if in a string literal.

```
package com.journaldev.java13.examples;
public class StringNewMethods {
        /***
         * New methods are to be used with Text Block Strings
         * @param args
         */
        @SuppressWarnings("preview")
        public static void main(String[] args) {

                String output = """
                            Name: %s
                            Phone: %d
                            Salary: $%.2f
                            """.formatted("Pankaj", 123456789, 2000.5555);

                System.out.println(output);


                String htmlTextBlock = "<html>   \n"+
                                              "\t<body>\t\t \n"+
                                                "\t\t<p>Hello</p>  \t \n"+
                                              "\t</body> \n"+
                                            "</html>";
                System.out.println(htmlTextBlock.replace(" ", "*"));
                System.out.println(htmlTextBlock.stripIndent().replace(" ", "*"));

                String str1 = "Hi\t\nHello' \" /u0022 Pankaj\r";
```

```
                    System.out.println(str1);
                    System.out.println(str1.translateEscapes());

          }

}
```

Output:

```
Name: Pankaj
Phone: 123456789
Salary: $2000.56

<html>***
          <body>                *
                    <p>Hello</p>**    *
          </body>*
</html>
<html>
          <body>
                    <p>Hello</p>
          </body>
</html>
Hi
Hello' " /u0022 Pankaj
Hi
Hello' " /u0022 Pankaj
```

☐ **Switch Expressions Enhancements – JEP 354**

Switch expressions were added as a preview feature in Java 12 release. It's almost same in Java 13 except that the "break" has been replaced with "yield" to return a value from the case statement.

```
package com.journaldev.java13.examples;
/**
 * JEP 354: Switch Expressions
 * https://openjdk.java.net/jeps/354
 * @author pankaj
 *
 */
public class SwitchEnhancements {

          @SuppressWarnings("preview")
          public static void main(String[] args) {
                    int choice = 2;

                    switch (choice) {
```

```
        case 1:
            System.out.println(choice);
            break;
        case 2:
            System.out.println(choice);
            break;
        case 3:
            System.out.println(choice);
            break;
        default:
            System.out.println("integer is greater than 3");
        }

        // from java 13 onwards - multi-label case statements
        switch (choice) {
        case 1, 2, 3:
            System.out.println(choice);
            break;
        default:
            System.out.println("integer is greater than 3");
        }

        // switch expressions, use yield to return, in Java 12 it was break
        int x = switch (choice) {
        case 1, 2, 3:
            yield choice;
        default:
            yield -1;
        };
        System.out.println("x = " + x);
    }

    enum Day {
        SUN, MON, TUE
    };

    @SuppressWarnings("preview")
    public String getDay(Day d) {
        String day = switch (d) {
        case SUN -> "Sunday";
        case MON -> "Monday";
        case TUE -> "Tuesday";
        };
        return day;
    }
}
```

- ☐ **FileSystems.newFileSystem() Method**

  Three new methods have been added to the FileSystems class to make it easier to use file system providers, which treats the contents of a file as a file system.

  newFileSystem(Path)
  newFileSystem(Path, Map<String, ?>)
  newFileSystem(Path, Map<String, ?>, ClassLoader)

- ☐ **DOM and SAX Factories with Namespace Support**

  There are new methods to instantiate DOM and SAX factories with Namespace support.

  newDefaultNSInstance()
  newNSInstance()
  newNSInstance(String factoryClassName, ClassLoader classLoader)

- ☐ **Reimplement the Legacy Socket API – JEP 353**
- ☐ **Dynamic CDS Archive – JEP 350**
- ☐ **ZGC: Uncommit Unused Memory – JEP 351**
- ☐ **Support for Unicode 12.1**