Java New Version Features from 10 to 20:

JAVA 10:

1. Local Variable Type Inference
2. API Changes in List, Set and Map [copyOf()]
3. API Changes in Collectors class in Streams

JAVA 11:

1. Executing a Java application by using a single command.
2. New Methods in String class.
3. Local Variable Type inference for the Lambda Expression parameters.
4. Accessing Private members from the Nested Classes Enhancement.
5. Writing String data to the files and Reading String data from the Files Enhancement.

Local Variable's Type Inference:

Up to Java 9 version, we are able to declare the local variables in the following syntax.

DataType varName = value;

EX:
int a = 10;

In JAVA 10 version, it is not required to specify the data type in the variables declaration, we can provide the keyword 'var' in place of Data Type , in this context the compiler and JVM will identify the data type to the variable on the basis of the right side value, this feature is called Type inference.

Syntax:
var varName = 10;

EX:
var a = 10;

EX:

```
public class Main {
   public static void main(String[] args) {
       int a = 10;
       System.out.println(a);
```

```
        var b = 20;
        System.out.println(b);
    }
}
```

```
10
20
```

In Java applications, we are able to apply type inference to the loop variables also.

EX:

```
public class Main {
    public static void main(String[] args) {
        for(int i = 0; i < 10; i++){
            System.out.println(i);
        }
        System.out.println();

        for(var i = 0; i < 10; i++){
            System.out.println(i);
        }
    }
}
```

```
0
1
2
3
4
5
6
7
8
9

0
1
2
3
4
5
6
```

```
7
8
9
```

In Java applications, we are able to apply type inference to the Arrays also, but we have to use the following syntax.

var refVar = new DataType[]{val_1, val_2,.... Val_n};
EX:

```java
public class Main {
    public static void main(String[] args) {
        int[] intArray = {10,20,30,40,50};
        for(int index = 0; index< intArray.length; index++){
            System.out.println(intArray[index]);
        }
        System.out.println();

        var intArray1 = new int[]{10,20,30,40,50};
        for(var index = 0; index < intArray1.length; index++){
            System.out.println(intArray1[index]);
        }

    }
}
```

```
10
20
30
40
50

10
20
30
40
50
```

In Java applications, we are able to apply type inference for the collection variables also.

EX:
```
import java.util.ArrayList;

public class Main {
   public static void main(String[] args) {
       ArrayList<String> al = new ArrayList<String>();
       al.add("AAA");
       al.add("BBB");
       al.add("CCC");
       al.add("DDD");
       System.out.println(al);

       var al1 = new ArrayList<String>();
       al1.add("AAA");
       al1.add("BBB");
       al1.add("CCC");
       al1.add("DDD");
       System.out.println(al1);



   }
}
```

```
[AAA, BBB, CCC, DDD]
[AAA, BBB, CCC, DDD]
```

In general, in Collections, we are able to use Diamond operator at right side part of the Collection object creation statement, where Diamond operator will take the type from left side part of expression, but in the case of Type inference no data type is provided at left side , in this context, JVM will provide Object type as Generic Type for the Collection, here the Collection object is able to allow any type of data.

EX:
```
import java.util.ArrayList;

public class Main {
   public static void main(String[] args) {
       ArrayList<String> al = new ArrayList<>();
       al.add("AAA");
```

```
        al.add("BBB");
        al.add("CCC");
        al.add("DDD");
        System.out.println(al);

        var al1 = new ArrayList<>();
        al1.add("AAA");
        al1.add("BBB");
        al1.add("CCC");
        al1.add("DDD");
        al1.add(10);
        al1.add(22.22);
        System.out.println(al1);



    }
}
```

```
[AAA, BBB, CCC, DDD]
[AAA, BBB, CCC, DDD, 10, 22.22]
```

Limitations to Type Inference:
1. Type inference is applicable for only Local variables, it is not applicable to the class level variables.

```
public class Main {
    //var b = 20; ---> Error
    public static void main(String[] args) {
        var a = 10;
        System.out.println(a);
    }
}
```

2. Type inference is not applicable for the methods which are having no value and null value.

```
public class Main {
    public static void main(String[] args) {
        var a = 10;
        System.out.println(a);

        String str = null;
        //var str1 = null; ----> Error

    }
}
```
10

3. Type INference is not applicable for the lambda expression parameters, but Type inference is applicable for the local variables inside the Lambda expressions.

EX:
```
interface I{
   int add(int a, int b);
}
public class Main {
   public static void main(String[] args) {
      /* I i = (var a, var b) -> {     Error
           return a+b;
       };*/

       I i = (a,b)->{
         var result = a+b;
         return result;
       };
       System.out.println(i.add(10,20));
   }
}
```

30

API Changes in List, Set and Map:

To make a List or a Set or a Map as Unmodifiable List, Set and Map JAVA 10-version has provided the following method from List, Set and Map.

public static List copyOf(List l)
public static Set copyOf(Set s)
public static Map copyOf(Map m)

EX:
```
import java.util.*;

public class Main {
   public static void main(String[] args) {
       var list = new ArrayList<>();
       list.add("AAA");
       list.add(10);
       list.add("22.22f");
       System.out.println(list);

       var list1 = List.copyOf(list);
       System.out.println(list1);
       //list1.add("BBB");-->
java.lang.UnsupportedOperationException

       var set = new HashSet<>();
       set.add("AAA");
       set.add(20);
       set.add(33.33);
       System.out.println(set);
       var set1 = Set.copyOf(set);
       System.out.println(set1);
       //set1.add("BBB"); --->
java.lang.UnsupportedOperationException

       var map = new HashMap<>();
       map.put(1, 111);
       map.put("A", "AAA");
       map.put(2,222);
       System.out.println(map);
       var map1 = Map.copyOf(map);
       System.out.println(map1);
       //map1.put("B", "BBB");
java.lang.UnsupportedOperationException
   }
}
```

```
[AAA, 10, 22.22f]
[AAA, 10, 22.22f]
```

```
[AAA, 20, 33.33]
[AAA, 20, 33.33]
{1=111, A=AAA, 2=222}
{A=AAA, 2=222, 1=111}
```

API Changes in Collectors class:

In Stream API, Collectors class has toList(), toSet() and toMap() method to generate List, Set and Map objects with the elements, here the generated List, Set and Map objects are not unmodifiable Collections, we can perform modifications over the List, Set and Map, but JAVA 10 version has provided the following static methods in Collectors class to generate Unmodifiable List, Unmodifiable Set and Unmodifiable Map.

```java
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class Main {
   public static void main(String[] args) {
       List<String> list = Arrays.asList("AAA", "BBB", "CCC",
"DDD");
       System.out.println(list);
       List<String> list1 = list.stream().map(str-
>str.toLowerCase()).collect(Collectors.toList());
       System.out.println(list1);
       list1.add("eee");
       System.out.println(list1);
       System.out.println();

       var list2 = Arrays.asList("AAA", "BBB", "CCC", "DDD");
       System.out.println(list2);
       var list3 = list2.stream().map(str-
>str.toLowerCase()).collect(Collectors.toUnmodifiableList());
       System.out.println(list3);
       //list3.add("eee"); --->
java.lang.UnsupportedOperationException
```

```
    }
}
```

```
[AAA, BBB, CCC, DDD]
[aaa, bbb, ccc, ddd]
[aaa, bbb, ccc, ddd, eee]

[AAA, BBB, CCC, DDD]
[aaa, bbb, ccc, ddd]
```

EX:

```
import java.util.Arrays;
import java.util.Set;
import java.util.stream.Collectors;

public class Main {
    public static void main(String[] args) {
        var set = Set.of(10,20,30,40,50);
        System.out.println(set);
        var set1 = set.stream().map(num-
>2*num).collect(Collectors.toSet());
        System.out.println(set1);
        set1.add(120);
        System.out.println(set1);
        System.out.println();

        var set2 = set.stream().map(num-
>2*num).collect(Collectors.toUnmodifiableSet());
        System.out.println(set2);
        //set2.add(120);--->
java.lang.UnsupportedOperationException
    }
}
```

Java 11:

1. Executing a Java application by using a single command.
2. New Methods in String class.
3. Local Variable Type inference for the Lambda Expression parameters.

4. Nested Base Access
5. Writing String data to the files and Reading String data from the Files Enhancement.

Download JAVA 11 version
https://jdk.java.net/java-se-ri/11-MR2

Executing a Java application by using a single command:
—---

Up to JAVA 10 version, we have to compile java files separately and we have to execute Java applications separately, for compiling a java file we have to use the "javac" command and for executing a java program we have to use the "java" command.

From JAVA 11 version onwards , no need to compile the java files separately by using javac command, we can compile java files and we can execute java applications directly by using a single "java" command.

Syntax:
```
java FileName.java
```

EX:
```
D:\java6\Test.java
class Test{
      public static void main(String[] args){
            System.out.println("Welcome To Java 11");
      }
}

D:\java6>set path=C:\java\jdk-10.0.2\bin;
D:\java6>java Test.java
Error: Could not find or load main class Test.java

D:\java6>set path=C:\java\jdk-11.0.0.1\bin;
D:\java6>java Test.java
Welcome To Java 11
```

If we execute the java file like above , the compiler will compile the java file and the compiler will generate .class files on the basis of the classes, abstract classes, interfaces, enums and inner classes.

After generating the .class files the JVM will search for the main() method in all the classes in an order in which we provided all the classes in the java file, if any class is identified with moain() method then the JVM will execute that main() method.

```
EX: D:\java6\Test.java
class A{
      public static void main(String[] args){
            System.out.println("main()-A");
      }
}

class B{
      public static void main(String[] args){
            System.out.println("main()-B");
      }
}

class C{
      public static void main(String[] args){
            System.out.println("main()-C");
      }
}
```

```
D:\java6>java Test.java
main()-A

D:\java6>
```

```
EX: D:\java6\Test.java
class B{
      public static void main(String[] args){
            System.out.println("main()-B");
      }
}

class C{
      public static void main(String[] args){
            System.out.println("main()-C");
      }
}

class A{
      public static void main(String[] args){
            System.out.println("main()-A");
      }
}


D:\java6>java Test.java
main()-B

D:\java6>
```

EX: D:\java6\Test.java

```
class C{
      public static void main(String[] args){
            System.out.println("main()-C");
      }
}

class A{
      public static void main(String[] args){
            System.out.println("main()-A");
      }
}

class B{
      public static void main(String[] args){
            System.out.println("main()-B");
      }
}
```

```
D:\java6>java Test.java
main()-C

D:\java6>
```

New Methods in String class:

To improve String class capabilities, JAVA 11 version has provided the following new methods in String class.

1. public String repeat(int count)

2. public boolean isBlank()
3. public String strip()
4. public String stripLeading()
5. public String stripTrailing()

public String repeat(int count)
It is able to perform the concatenation over the provided String up to the specified number of times.
EX:
```java
public class Main {
   public static void main(String[] args) {
       String data = new String("RAMA ");
       System.out.println(data);
       String newData = data.repeat(10000000);
       System.out.println(newData);
   }
}
```

public boolean isBlank():
This method is able to check whether the String is blank or not, if the String is blank then it will return true value otherwise it will return false value.

Q)What is the difference between isEmpty() and isBlank() methods in String class?

Ans:
----

1. isEmpty() was provided in JDk 1.6 version.
   isBlank() was provided in JDK 11 version.

2. isEmpty() method will check whether the String data is empty or not, if the String data is empty then it will return true otherwise it will return false, if the String contains spaces then it will return false value.

   isBlank() method will check whether the String data is empty or not, if the String data is empty then it will return true value otherwise it will return false value, if the String data contains spaces then it will return true value.

EX:
```java
public class Main {
   public static void main(String[] args) {
       String data1 = "";
       System.out.println(data1.isBlank());
       System.out.println(data1.isEmpty());

       String data2 = "      ";
       System.out.println(data2.isBlank());
       System.out.println(data2.isEmpty());
   }
}
```
```
true
true
true
false
```

public String strip():
It is the same as the trim() method, it will remove the spaces before the String and after the String.

Public String stripLeading():
It will remove all the spaces before the String data.

public String stripTrailing():
It will remove all the spaces after the String data

EX:
```java
public class Main {
   public static void main(String[] args) {
       String data = "      Durga Software Solutions        ";
       System.out.println(data);
       System.out.println(data.strip());
       System.out.println(data.stripLeading());
       System.out.println(data.stripTrailing());
   }
}
```
```
      Durga Software Solutions
Durga Software Solutions
Durga Software Solutions
Durga Software Solutions
```

Local Variable Type inference for the Lambda Expression parameters:
Up to JAVA 10 version, type inference is not possible for the lambda expression parameters, but from JAVA 11 version onwards Type inference is possible for the Lambda expression parameters.

```
interface Calculator{
   int sqr(int num);
}
public class Main {
   public static void main(String[] args) {
       Calculator calculator = (var num) -> num*num;
       System.out.println(calculator.sqr(4));
   }
}
```

```
16
```

Nested Base Access:

Up to Java 10 version, in inner classes, if we access private member of any inner class in the respective outer class by using reflection API then we are able to get an exception like java.lang.IllegalAccessException, but in JAVA 11 version we will not get any exception, we are able to access the private member and we will get output.

EX:
```
import java.lang.reflect.Field;

public class Main {
   class Test{
       private int count = 10;
   }
   public static void main(String[] args) throws
NoSuchFieldException, IllegalAccessException {
       Main.Test mt = new Main().new Test();
       System.out.println(mt.count);

       Field field = mt.getClass().getDeclaredField("count");
       field.setInt(mt, 2000);
       System.out.println(mt.count);
```

```
    }
}
```

```
10
2000
```

Writing String data to the files and Reading String data from the Files:

JAVA 11 version has provided the following 4 new methods in java.nio.file.Files class.

Public static Path writeString(Path path, CharSequence csq, OpenOption … options)

Public static Path writeString(Path path, CharSequence csq, Charset cs OpenOption … options)

public static String readString(Path path)

Public static String readString(Path p, CharSet cs)

EX:

```
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;

public class Main {
   public static void main(String[] args)throws Exception{
       Path path = Paths.get("E:/","abc/xyz", "welcome.txt");
       Files.writeString(path, "Welcome To Durga Software 
Solutions");
       String data = Files.readString(path);
       System.out.println(data);
   }
}
```

JAVA 12 Version Features:
----
1. Switch Enhancements
2. Files mismatch() Method
3. Compact Number Formatting
4. Teeling Collectors in Stream API

5. Java String class new Methods

---
1. Switch Enhancements:
In Java 12 Switch Enhancements are provided as Preview features only, it is not standardized , in JAVA 14 version it was provided as a standard implementation.

2. Files mismatch() Method:

It can be used to check whether two files content is mismatched or not, if the two files content is matched then the mismatch() method will return -1L, if the two files content is mismatched then the mismatch() method will return +ve value.

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

public class Main {
   public static void main(String[] args) throws IOException {
       Path path1 = Paths.get("E:/","abc/xyz","welcome1.txt");
       Path path2 = Paths.get("E:/","abc/xyz","welcome2.txt");
       Path path3 = Paths.get("E:/","abc/xyz","welcome3.txt");

       Files.writeString(path1, "Welcome To Durga Software
Solutions");
       Files.writeString(path2, "Welcome To Durga Software
Solutions");
       Files.writeString(path3, "Welcome To Durgasoft");

       long val1 = Files.mismatch(path1, path2);
       System.out.println(val1);
       if(val1 == -1L){
           System.out.println("Files welcome1.txt, welcome2.txt
Content Is Matched");
       }else{
           System.out.println("Files welcome1.txt, welcome2.txt
Content Is Mismatched");
       }

       System.out.println();
```

```
        long val2 = Files.mismatch(path1, path3);
        System.out.println(val2);
        if(val2 == -1L){
            System.out.println("Files welcome1.txt, welcome3.txt
Content Is Matched");
        }else{
            System.out.println("Files welcome1.txt, welcome3.txt
Content Is Mismatched");
        }




    }
}
```

```
-1
Files welcome1.txt, welcome2.txt Content Is Matched

16
Files welcome1.txt, welcome3.txt Content Is Mismatched
```

Compact Number Formatting:
In general, in java applications, we are able to represent numbers in numeric form, not in the short forms like 1k, 10k,... , and the long forms like 1 Thousand, 10 Thousand,... to represent numbers in Short form and Long form JAVA 12 version has provided a predefined class in the form of java.text.CompactNumberFormat.

IN Java applications, by using the CompactNumberFormat class we are able to perform the following conversions.

1. Converting Numbers from Normal representation to Long Form and Short Form Representations:
   100 -------> 100
   1000 ------> 1 Thousand or 1 k
   10000 -----> 10 Thousand or 10 k

2. Converting Numbers from Short or Long form to normal representations:

```
      100 -------> 100
      1k --------> 1000
      10k -------> 10000

      100 -------> 1000
      1 Thousand ---> 1000
      10 Thousand --> 10000
```

To perform the above conversions, first we have to get the COmpactNumberFormat object.

```
CompactNumberFormat cnf = CompactNumberFormat.getCompactNumberInstance(Locale
l,  COmpactNumberFormat.style.LONG);
```

To convert the number from normal representation to compact number representation we have to use the following method.

```
public String format(Number num)
```

To Convert Compact Number to normal representation we have to use the following method.

```
public Number parse(String compactNumber)
```

EX:

```
import java.text.NumberFormat;
import java.util.Locale;

public class Main {
   public static void main(String[] args){
       NumberFormat numberFormat =
NumberFormat.getCompactNumberInstance(new Locale("en",
"US"),NumberFormat.Style.SHORT);
       System.out.println("1000 ------->
"+numberFormat.format(1000));
       System.out.println("10000 ----->
"+numberFormat.format(10000));
       System.out.println("100000 ---->
"+numberFormat.format(100000));

   }
}
```

```
1000 ------> 1K
10000 -----> 10K
100000 ----> 100K
```

EX:

```java
import java.text.NumberFormat;
import java.util.Locale;

public class Main {
   public static void main(String[] args){
       NumberFormat numberFormat =
NumberFormat.getCompactNumberInstance(new Locale("en",
"US"),NumberFormat.Style.LONG);
       System.out.println("1000 ------>
"+numberFormat.format(1000));
       System.out.println("10000 ----->
"+numberFormat.format(10000));
       System.out.println("100000 ---->
"+numberFormat.format(100000));

   }
}
```

```
1000 ------> 1 thousand
10000 -----> 10 thousand
100000 ----> 100 thousand
```

EX:

```java
import java.text.NumberFormat;
import java.text.ParseException;
import java.util.Locale;

public class Main {
   public static void main(String[] args) throws ParseException {
       NumberFormat numberFormat =
NumberFormat.getCompactNumberInstance(new Locale("en",
"US"),NumberFormat.Style.LONG);
       System.out.println("1 thousand ------>
"+numberFormat.parse("1 thousand"));
       System.out.println("10 thousand ----->
"+numberFormat.parse("10 thousand"));
       System.out.println("10 thousand ---->
"+numberFormat.parse("100 thousand"));
```

```
    }
}
```

```
1 thousand ------> 1000
10 thousand -----> 10000
100 thousand ----> 100000
```

EX:

```
import java.text.NumberFormat;
import java.text.ParseException;
import java.util.Locale;

public class Main {
    public static void main(String[] args) throws ParseException {
        NumberFormat numberFormat =
NumberFormat.getCompactNumberInstance(new Locale("en",
"US"),NumberFormat.Style.SHORT);
        System.out.println("1k ------> "+numberFormat.parse("1K"));
        System.out.println("10k -----> "+numberFormat.parse("10K"));
        System.out.println("100k ---->
"+numberFormat.parse("100K"));

    }
}
```

```
1k ------> 1000
10k -----> 10000
100k ----> 100000
```

Teeling Collectors in Stream API:

The main purpose of Teeing Collectors in Stream API is to take two streams as an input , performing BIFunction and return the generated results.

Public static Collector teeing(Collector stream1, Collector strem2, BIFunction merger)

EX:

```
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class Main {
   public static void main(String[] args){
       double mean =
Stream.of(1,2,3,4,5,6,7,8,9,10).collect(Collectors.teeing(
               Collectors.summingDouble(x->x),
               Collectors.counting(),
               (sum,count)->sum/count));
       System.out.println(mean);

   }
}
```

```
5.5
```

5. Java String class new Methods

JAVA 12 version has provided the following new methods in String class.

1. indent()
2. transform()

public String indent(int count):
The main purpose of this method is to add spaces or to remove spaces from a String.

If count < 0 Then the spaces will be removed at the beginning of each and every line.

If count > 0 then spaces will be added at the beginning of String.

If -count > existed spaces then all spaces will be removed.

EX:

```
public class Main {
   public static void main(String[] args){
       String str1 = "Durga\nSoftware\nSolutions";
       System.out.println(str1);
       String str2 = str1.indent(5);
       System.out.println(str2);
```

```
        String str3 = new String("      Durga\n      Software\n
Solutions");
        System.out.println(str3);
        String str4 = str3.indent(-5);
        System.out.println(str4);

        String str5 = new String("  Durga\n  Software\n
Solutions");
        System.out.println(str5);
        String str6 = str5.indent(-5);
        System.out.println(str6);
    }
}
```

```
Durga
Software
Solutions
     Durga
     Software
     Solutions

     Durga
     Software
     Solutions
Durga
Software
Solutions

  Durga
  Software
  Solutions
Durga
Software
Solutions
```

public R transform(Function f):
It will take a Function as parameter and it will transform a String in to some other form and it will return the generated results.

EX:

```
public class Main {
   public static void main(String[] args){
       String data = "Durga Software Solutions";
       System.out.println(data);
       String newData = data.transform(str->str.toUpperCase());
       System.out.println(newData);
   }
}
```

```
Durga Software Solutions
DURGA SOFTWARE SOLUTIONS
```

JAVA 13 and 14 Features:
----

1. Text Blocks[Preview]
2. New Methods in String class for Text blocks[Preview]
3. Switch Enhancements [13.Preview, 14. Standard]
4. Pattern Matching for instanceof operator[Preview]
5. NullPointerException enhancement[14]
6. Records[Preview]

Switch Enhancements:

Up to JAVA 13 version switch is unable to return a value, but the new enhancement for switch in JAVA 14 version is able to return a value.

In Java 14 version, switch is able to return a value in the following two ways.

1. By Using ‘yield’ statement.
2. By Using Lambda Style.

Returning a value by Using yield statement:

```
var val = switch(paramValue){
case 1:
      yield value1;
case 2:
      yield value2;
-----
-----
case n:
      yield val_n;
default:
      yield val_default;
}
```

EX:

```java
public class Main {
    public static void main(String[] args){
        int dayNo = 7;
        String dayName = switch (dayNo){
            case 1:
                yield "MONDAY";
            case 2:
                yield "TUESDAY";
            case 3:
                yield "WEDNESDAY";
            case 4:
                yield "THURSDAY";
            case 5:
                yield "FRIDAY";
            case 6:
                yield "SATURDAY";
            case 7:
                yield "SUNDAY";
            default:
                yield "No Day Name is defined with the dayNo";
        };

        System.out.println(dayName);

    }
}
```

By Using Lambda Style:

```
var value = switch(paramVal){
      case 1 -> val_1;
      case 2 -> val_2;
      ----
      ----
      case n -> val_n;
      default -> value_default;
```

}

EX:
```java
public class Main {
   public static void main(String[] args){
       int dayNo = 5;
       String dayName = switch (dayNo){
           case 1 -> "MONDAY";
           case 2 -> "TUESDAY";
           case 3 -> "WEDNESDAY";
           case 4 -> "THURSDAY";
           case 5 ->  "FRIDAY";
           case 6 ->  "SATURDAY";
           case 7 ->  "SUNDAY";
           default ->  "No Day Name is defined with the dayNo";
       };

       System.out.println(dayName);

   }
}
```

Java 14 version is able to allow multi labeled cases, that is more than one case label with a single case.

EX:
```java
public class Main {
   public static void main(String[] args){
       int dayNo = 7;
       String dayName = switch (dayNo){
           case 1,2,3,4,5 -> "WEEKDAY";

           case 6,7 ->  "WEEKEND";

           default ->  "No is not representing WEEKDAY and 
WEEKEND";
       };

       System.out.println(dayName);

   }
}
```

EX:

```java
public class Main {
   public static void main(String[] args){
       int dayNo = 10;
       String dayName = switch (dayNo){
           case 1,2,3,4,5:
               yield "WEEKDAY";

           case 6,7:
               yield "WEEKEND";

           default:
               yield "No is not representing WEEKDAY and WEEKEND";
       };

       System.out.println(dayName);

   }
}
```

NullpointerException Enhancement:
In Java 13 version, NullpointerException has provided no exception description, but in JAVA 14 version NullPointerException has a meaningful Exception description.

```java
import java.util.Date;

public class Main {
   public static void main(String[] args){
       Date date = null;
       System.out.println(date.toString());

   }
}
```

```
PS D:\java6\intellij Idea\app14\src> javac Main.java
PS D:\java6\intellij Idea\app14\src> java -
XX:+ShowCodeDetailsInExceptionMessages Main
Exception in thread "main" java.lang.NullPointerException: Cannot invoke
"java.util.Date.toString()" because "<local1>" is null
        at Main.main(Main.java:6)
```

Java Features from 15 to 20:

----
1. Text Blocks
2. Pattern Matching for instanceof operator
3. Records

Text Blocks:

Up to Java 14 version , if we want to write String data in more than one line then we have to use \n character in the middle of the String.

```
public class Main {
   public static void main(String[] args) {
       String address = "\tDurga Software Solutions\n"+
               "\t202, HMDA, Mitrivanam\n"+
               "\tAmeerpet, Hyderabad-38";
       System.out.println(address);
   }
}
```

```
      Durga Software Solutions
      202, HMDA, Mitrivanam
      Ameerpet, Hyderabad-38
```

In the above code, to bring data to the new line we have to use \n character and to provide space before each and every line we have to use \t, these characters are called Escape characters and these escape characters may or may not be supported by the operating systems.

To overcome the above problem we have to use Text blocks provided by Java in its latest version.

In Text Blocks we will data in multiple lines without using \n character and \t characters.
Syntax:
String varName = “””
String data in Line-1
String data in Line-2

“”” ;

EX:

```
public class Main {
   public static void main(String[] args) {
       String address = """
                   Durga Software Solutions
                   202, HMDA, Mitrivanam
                   Ameerpet, Hyderabad-38
               """;
       System.out.println(address);
   }
}
```

```
    Durga Software Solutions
    202, HMDA, Mitrivanam
    Ameerpet, Hyderabad-38
```

In Text Block, we have to provide data in the next line after “”” and we can end the text block by providing ""” in the same line of the text block or in the next line.

EX:

```
String address = """Durga Software Solutions
           202, HMDA, Mitrivanam
           Ameerpet, Hyderabad-38
       """;
```

Status: Compilation Error

EX:

```
String address = """
       Durga Software Solutions
       202, HMDA, Mitrivanam
       Ameerpet, Hyderabad-38""";
```

Status: No Compilation Error

In Java applications, if we provide the same data in “” and in “”” “”” then we are able to compare these two String data by using equals() method and by using == operator.

```
public class Main {
   public static void main(String[] args) {
       String str1 = "Durga Software Solutions";
       String str2 = """
               Durga Software Solutions""";
       System.out.println(str1);
```

```
        System.out.println(str2);
        System.out.println(str1 == str2);
        System.out.println(str1.equals(str2));
    }
}
```

```
Durga Software Solutions
Durga Software Solutions
true
true
```

It is possible to perform Concatenation operation between the String data represented in the form of “” and “”” “””

```
public class Main {
    public static void main(String[] args) {
        String data1 = "Durga ";
        String data2 = """
                  Software""";
        String data3 = " Solutions";
        String result1 = data1+data2+data3;
        System.out.println(result1);
        String result2 = data1.concat(data2).concat(data3);
        System.out.println(result2);
    }
}
```

```
Durga Software Solutions
Durga Software Solutions
```

In Java applications, it is possible to pass text blocks as parameters to the methods and it is possible to return text blocks as return value from the methods.

```
class Student{
    public String getStudentDetails(String data){
        return """
                  Student Details
                  ----------------
                  """+data+ """
                  SEMAIL    :    durga@durgasoft.com
                  SMOBILE   :    9988776655
```

```java
                """;
    }
}
public class Main {
    public static void main(String[] args) {
        Student student = new Student();
        String studentData = student.getStudentDetails("""
                SID       :     S-111
                SNAME     :     Durga
                """);
        System.out.println(studentData);
    }
}
```

```
Student Details
----------------
SID      :    S-111
SNAME    :    Durga
SEMAIL   :    durga@durgasoft.com
SMOBILE  :    9988776655
```

In general, in Java applications, Text blocks are very much useful in representing SQL Queries, Html codes , JSON representations,....

EX:

```java
public class Main {
    public static void main(String[] args) {
        String query = """
                select
                    ENO,
                    ENAME,
                    ESAL,
                    EADDR
                from emp1
                """;
        System.out.println(query);
    }
}
```

```
select
    ENO,
    ENAME,
    ESAL,
    EADDR
from emp1
```

EX:

```
public class Main {
   public static void main(String[] args) {
       String html = """
               <html>
                   <body>
                       <h1>
                           Welcome To Durga Software Solutions
                       </h1>
                   </body>
               </html>
               """;
       System.out.println(html);
   }
}
```

```
<html>
    <body>
        <h1>
            Welcome To Durga Software Solutions
        </h1>
    </body>
</html>
```

EX:

```
public class Main {
   public static void main(String[] args) {
       String json = """
               {
                   "ENO": 111
                   "ENAME": "AAA"
                   "ESAL": 5000
                   "EADDR": Hyd
               }
               """;
       System.out.println(json);
```

```java
    }
}
```

```
{
    "ENO": 111
    "ENAME": "AAA"
    "ESAL": 5000
    "EADDR": Hyd
}
```

In Text Blocks if we want to provide spaces to the String in all lines then we have to use the following method.

public String indent(int spaces)

EX:

```java
public class Main {
    public static void main(String[] args) {
        String data = """
                  Durga Software Solutions
                  202, HMDA, Mitrivanam
                  Ameerpet, Hyderabad-38
                  """;
        System.out.println(data);
        System.out.println(data.indent(5));

    }
}
```

```
Durga Software Solutions
202, HMDA, Mitrivanam
Ameerpet, Hyderabad-38

     Durga Software Solutions
     202, HMDA, Mitrivanam
     Ameerpet, Hyderabad-38
```

It is possible to provide ‘’[Single quotations] , “”[Double quotations] and “””” “””” [Triple Quotations] with \ in the the text blocks

EX:

```java
public class Main {
    public static void main(String[] args) {
        String data = """
                'Durga Software Solutions'
                "Durga Software Solutions"
                \"""Durga Software Solutions\"""
                 """;
        System.out.println(data);


    }
}
```

```
'Durga Software Solutions'
"Durga Software Solutions"
"""Durga Software Solutions"""
```

public String formatted(Object … values)
This method can be used to format the data in text blocks.

EX:

```
public class Main {
    public static void main(String[] args) {
        String empData = """
                Employee Details
                -----------------
                Employee Number       : %d
                Employee Name         : %s
                Employee Salary       : %f
                Employee Address      : %s
                """;
        System.out.println(empData.formatted(111,"Durga",25000.0f,
"Hyderabad"));
        System.out.println(empData.formatted(222,"Nagoor",35000.0f,
"Chennai"));
        System.out.println(empData.formatted(333,"Rakesh",45000.0f,
"Delhi"));


    }
}
```

```
Employee Details
-----------------
Employee Number       : 111
Employee Name        : Durga
Employee Salary      : 25000.000000
Employee Address     : Hyderabad

Employee Details
-----------------
Employee Number      : 222
Employee Name        : Nagoor
Employee Salary      : 35000.000000
Employee Address     : Chennai

Employee Details
-----------------
Employee Number      : 333
Employee Name        : Rakesh
Employee Salary      : 45000.000000
Employee Address     : Delhi
```

Pattern Matching for instanceof operator:

Consider the following program.

```java
public class Main {
   public static void main(String[] args)  {
       Object data = "Durga Software Solutions";
       if(data instanceof String){
           String str = (String)data;
           str = str.toUpperCase();
           System.out.println(str);
       }else{
           System.out.println("Data does not have String
instance");
       }

   }
}
```

In the above program , after confirming the Data contains String instance, to perform String operations we have to perform Type casting from Object type to String type, In Java Latest version we can provide pattern matching with the instanceof operator directly without performing typecasting.

```java
public class Main {
    public static void main(String[] args) {
        Object data = "Durga Software Solutions";
        if(data instanceof String str){
            System.out.println(str.toUpperCase());
        }else{
            System.out.println("Data does not have String
instance");
        }
    }
}
```

DURGA SOFTWARE SOLUTIONS

Records:

In general, in Java applications we are able to write data carriers like java bean classes, where in Java bean classes we are able to provide the following conventions.

1. Private Properties
2. Public methods like Accessor methods and mutator methods.
3. As per the requirement we may declare equals(), hasCode() and toString(),....

EX:

```
class Employee{
    private int eno;
    private String ename;
    private float esal;
    private String eaddr;

    public Employee(int eno, String ename, float esal, String eaddr)
{
        this.eno = eno;
        this.ename = ename;
        this.esal = esal;
        this.eaddr = eaddr;
    }

    public int getEno() {
        return eno;
    }

```

```
   public void setEno(int eno) {
       this.eno = eno;
   }

   public String getEname() {
       return ename;
   }

   public void setEname(String ename) {
       this.ename = ename;
   }

   public float getEsal() {
       return esal;
   }

   public void setEsal(float esal) {
       this.esal = esal;
   }

   public String getEaddr() {
       return eaddr;
   }

   public void setEaddr(String eaddr) {
       this.eaddr = eaddr;
   }
}
public class Main {
   public static void main(String[] args) {
       Employee employee = new Employee(111,"Durga", 5000, "Hyd");
       System.out.println("Employee Details");
       System.out.println("-------------------------");
       System.out.println("Employee Number     : 
"+employee.getEno());
       System.out.println("Employee Name       : 
"+employee.getEname());
       System.out.println("Employee Salary     : 
"+employee.getEsal());
       System.out.println("Employee Address    : 
"+employee.getEaddr());
   }
}
```

```
Employee Details
------------------------
Employee Number     : 111
Employee Name       : Durga
Employee Salary     : 5000.0
Employee Address    : Hyd
```

In the above code , in Employee class we have provided a lot of boilerplate code like parameterized constructor, accessor methods, mutator methods,....

To remove the above boilerplate code Java has provided a new feature in the form of “Record” in latest versions.

Record is a simple data carrier, it will include properties, constructors, accessor methods, toString(),hashCode(), equals(),....

Syntax:

```
record Name(ParameterList){
}
```

EX:

```
record Employee(int eno, String ename, float esal, String eaddr){
}
```

If we compile the record then the compiler will translate the record into the final class like below.

```
final class Employee extends java.lang.Record {
  Employee(int, java.lang.String, float, java.lang.String);
  public final java.lang.String toString();
  public final int hashCode();
  public final boolean equals(java.lang.Object);
  public int eno();
  public java.lang.String ename();
  public float esal();
  public java.lang.String eaddr();
}
```

Conclusions:

1. Every record must be a final class internally, it is not possible to define inheritance relation between records.

2. Every record should be a child class to java.lang.Record, where java.lang.Record will provide the methods like hashcode(), equals(), toString() methods.

```
public abstract class java.lang.Record {
  protected java.lang.Record();
  public abstract boolean equals(java.lang.Object);
  public abstract int hashCode();
  public abstract java.lang.String toString();
}
```

3. Every record contains a parameterized constructor called Canonical Constructor, where the parameters of the canonical constructors are the same as the parameter types which we provided to the records.

4. In records, all the parameters are converted as private and final , we are unable to access them outside of the record and we are unable to modify their values.

5. In record, for each and every property a separate accessor method is provided , where the accessor method names are the same as the property name not like getXXX() methods, like propertyName().

6. In records, toString() method was implemented in such a way that to return a string that contains “RecordName[Prop1=val1,prop2=val2,prop3=val3…]”

7. In Records, equals() method was implemented in such a way that to perform Content comparison instead of references comparison.

Note: IN JAVA / J2EE applications we will use records as an alternative to the Java bean classes.

EX:

```
record Employee(int eno, String ename, float esal, String eaddr){
}
public class Main {
   public static void main(String[] args) {
       Employee employee = new Employee(111, "Durga", 50000,
"Hyderabad");
       System.out.println("Employee Details");
       System.out.println("--------------------");
       System.out.println("Employee Number       :
"+employee.eno());
```

```
       System.out.println("Employee Name        : 
"+employee.ename());
       System.out.println("Employee Salary      : 
"+employee.esal());
       System.out.println("Employee Address     : 
"+employee.eaddr());
   }
}
```

```
Employee Details
--------------------
Employee Number      : 111
Employee Name        : Durga
Employee Salary      : 50000.0
Employee Address     : Hyderabad
```

EX:

```
record Employee(int eno, String ename, float esal, String eaddr){
}
public class Main {
   public static void main(String[] args) {
       Employee employee1 = new Employee(111, "Durga", 50000, 
"Hyderabad");
       Employee employee2 = new Employee(222, "Nagoor", 60000, 
"Chennai");
       Employee employee3 = new Employee(111, "Durga", 50000, 
"Hyderabad");

       System.out.println(employee1);
       System.out.println(employee2);
       System.out.println(employee3);

       System.out.println(employee1.equals(employee2));
       System.out.println(employee1.equals(employee3));
   }
}
```

```
Employee[eno=111, ename=Durga, esal=50000.0, eaddr=Hyderabad]
Employee[eno=222, ename=Nagoor, esal=60000.0, eaddr=Chennai]
Employee[eno=111, ename=Durga, esal=50000.0, eaddr=Hyderabad]
false
true
```

In records, we are able to declare our own variables in the body, but the variables must be static variables, because record is not allowing instance variables.

EX:

```
record Employee(int eno, String ename, float esal, String eaddr){
   static String eemail;
   static String emobile;

   public static String eemail() {
       return eemail;
   }

   public static void setEemail(String eemail) {
       Employee.eemail = eemail;
   }

   public static String emobile() {
       return emobile;
   }

   public static void setEmobile(String emobile) {
       Employee.emobile = emobile;
   }
}
public class Main {
   public static void main(String[] args) {
       Employee employee = new Employee(111, "Durga", 50000,
"Hyd");
       Employee.setEemail("durga@dss.com");
       employee.setEmobile("91-9988776655");

       System.out.println("Employee Details");
       System.out.println("----------------------");
       System.out.println("Employee Number     : "+employee.eno());
       System.out.println("Employee Name       :
"+employee.ename());
       System.out.println("Employee Salary     :
"+employee.esal());
       System.out.println("Employee Address    :
"+employee.eaddr());
       System.out.println("Employee Email      :
"+employee.eemail());
```

```
       System.out.println("Employee Mobile     :
"+employee.emobile());
   }
}
```

```
Employee Details
----------------------
Employee Number     : 111
Employee Name       : Durga
Employee Salary     : 50000.0
Employee Address    : Hyd
Employee Email      : durga@dss.com
Employee Mobile     : 91-9988776655
```

In Records, we are able to manipulate accessor methods by writing explicitly.

```
record User(String fname, String lname, String address){
   @Override
   public String fname() {
       return "First Name      : "+fname;
   }

   @Override
   public String lname() {
       return "Last Name      : "+lname;
   }

   @Override
   public String address() {
       return "Address       : "+address;
   }
}
public class Main {
   public static void main(String[] args) {
       User user = new User("Durga", "N", "Hyderabad");
       System.out.println("User Details");
       System.out.println("------------------");
       System.out.println(user.fname());
       System.out.println(user.lname());
       System.out.println(user.address());
   }
}
```

```
User Details
------------------
First Name      : Durga
Last Name      : N
Address       : Hyderabad
```

In Records, we are able to declare our own constructors with the following conditions.

1. First we must define Canonical constructor explicitly.
2. Prepare our own constructor, but our own constructor must access the canonical constructor by using this() .

EX:

```java
record Employee(int eno, String ename, float esal, String eaddr){
   Employee(int eno, String ename, float esal, String eaddr) {
       this.eno = eno;
       this.ename = ename;
       this.esal = esal;
       this.eaddr = eaddr;
   }

   public Employee(){
       this(111,"AAA",5000,"Hyd");
   }

   public Employee(int eno, String ename, float esal) {
       this(eno, ename, esal, "Hyd");
   }
}
public class Main {
   public static void main(String[] args) {
       Employee employee1 = new Employee();
       System.out.println(employee1);

       Employee employee2 = new Employee(111,"Durga", 5000);
       System.out.println(employee2);

       Employee employee3 = new Employee(111, "Durga", 5000,
"Hyd");
       System.out.println(employee3);
   }
}
```

```
Employee[eno=111, ename=AAA, esal=5000.0, eaddr=Hyd]
Employee[eno=111, ename=Durga, esal=5000.0, eaddr=Hyd]
Employee[eno=111, ename=Durga, esal=5000.0, eaddr=Hyd]
```

In the above example we are able to access one constructor from another constructor in the record, so Records are following Constructor chaining.

In Records, if we declare a 0-arg constructor without any parameter and without () then that constructor is called a compact constructor, it will be executed just before executing the canonical constructor and it will be used for the data validations.

EX:

```java
record A(int i, int j){
   public A{
       System.out.println("Compact Constructor");
   }
   public void add(){
       System.out.println("ADD    : "+(i+j));
   }
}
public class Main {
   public static void main(String[] args) {
       A a = new A(10,20);
       a.add();
   }
}
```

```
Compact Constructor
ADD   : 30
```

EX:

```java
record User(String uname, String upwd){
   public User{
       if(uname == null || uname.equals("")){
           System.out.println("User Name is Required.");
       }
       if(upwd == null || upwd.equals("")){
           System.out.println("User Password is Required.");
       }
   }
   public void getUserDetails(){
       System.out.println("User Details");
       System.out.println("----------------");
       System.out.println("User Name     : "+uname);
       System.out.println("Password      : "+upwd);
   }
}
public class Main {
   public static void main(String[] args) {
       User user = new User("","");
       user.getUserDetails();
       System.out.println();

       User user1 = new User("durga", "durga123");
       user1.getUserDetails();

   }
}
```

```
User Name is Required.
User Password is Required.
User Details
----------------
User Name    :
Password     :

User Details
----------------
User Name    : durga
Password     : durga123
```

IN Java applications, we are unable to extend one record to another record , but we are able to implement interfaces in the record.

EX:

```
import java.io.Serializable;

interface Car{
    public void getCarDetails();
}
record FordCar(String model, String type, int price) implements
Car, Serializable{

    @Override
    public void getCarDetails() {
```

```java
        System.out.println("Car Details");
        System.out.println("----------------");
        System.out.println("Car Model    : "+model());
        System.out.println("Car Type     : "+type());
        System.out.println("Car Price    : "+price());
    }
}
public class Main {
    public static void main(String[] args) {
        Car car = new FordCar("2015", "Echosport", 1200000);
        car.getCarDetails();
    }
}
```

```
Car Details
----------------
Car Model    : 2015
Car Type     : Echosport
Car Price    : 1200000
```