Functional programming:- (java 8 new features)

Functional programming imposes a declarative programming style, i.e we program through expressions or declarations (computation is expressed in terms of functions/expressions — black boxes(input->output)).

[Coding declaratively tends to better readable, and expressive code.]

Functional programming	1.Lambdas 2.method reference 3.functional interface 4.streams(filter-map reduce)

Lambda exp:

Lambda expressions only work with functional interfaces**.**

```java
Thread t = new Thread(new Runnable() {
  @Override
  public void run() {
    System.out.println("From a thread"
);
  }
});
```

```java
Thread t = new Thread(() -> System.out.println("From a threa
d"));
```

```java
Collections.sort(Arrays.asList(3, 2, 1
),
 new Comparator<Integer>() {
  @Override
  public int compare(Integer o1, Integ
er o2) {
    return o1.compareTo(o2);
  }
});
```

```java
Collections.sort(Arrays.asList(3, 2, 1), (o1, o2) -> o1.comp
areTo(o2));
```

```java
List<String> strings = Arrays.asList("
R", "O", "B");
for (String string : strings) {
  System.out.println(string);
}
```

```java
List<String> strings = Arrays.asList("R", "O", "B");
strings.forEach(s -> System.out.println(s));
// or use method reference if there is a
direct execution of a method on the source element
strings.forEach(System.out::println);
```

Method References

Lambda expressions are used to express a function's body, but **if you have the function/method already written/defined, you can directly use them as method references wherever lambda expressions could be used**.

1. **Reference to a static method** — ContainingClass::staticMethodName
2. **Reference to an instance method of a particular object** — containingObject::instanceMethodName
3. **Reference to an instance method of an arbitrary object of a particular type** _ ContainingType::methodName
4. **Reference to a constructor** — ClassName::new

```java
public class MethodRef {
  public void printLowerCase(String s) {
    System.out.println(s.toLowerCase());
  }
  public static void printUpperCase(String s) {
    System.out.println(s.toUpperCase());
  }
  public void publicMethod() {
    List<String> list = Arrays.asList("A", "B", "C");
    list.forEach(MethodRef::printUpperCase); // Reference to a static method
    list.forEach(this::printLowerCase);   // Reference to an instance method of a particular object
    list.forEach(String::toLowerCase); // instance method of an object of a particular type
    list.forEach(String::new);          // Reference to a constructor
  }
}
```

Functional Interfaces

A Functional Interface is an **interface in Java that has only one abstract method**

```java
interface Runnable
{
    Public void run();
}
```

Streams

Streams in Java allow for defining a pipeline of operations that can carry out the transformation of input data into the required form.

Source

- **Collection.stream()** — Create a stream from the elements of a collection
- **Stream.of(T...)** — Create a stream from the arguments passed to the factory method
- **Stream.of(T[])** — Create a stream from the elements of an array
- **Stream.empty()** — Create an empty stream
- **IntStream.range(lower, upper)** — Create an IntStream consisting of the elements from lower to upper, exclusive
- **IntStream.rangeClosed(lower, upper)** — Create an IntStream consisting of the elements from lower to upper, inclusive.

intermediate operations

- **filter(Predicate<T>)** — The elements of the stream matching the predicate
- **map(Function<T, U>)** — The result of applying the provided function to the elements of the stream
- **flatMap(Function<T, Stream<U>>** — The elements of the streams resulting from applying the provided stream-bearing function to the elements of the stream
- **distinct()** — The elements of the stream, with duplicates removed
- **sorted()** — The elements of the stream, sorted in natural order
- **Sorted(Comparator<T>)** — The elements of the stream, sorted by the provided comparator
- **limit(long)** — The elements of the stream, truncated to the provided length
- **skip(long)** — The elements of the stream, discarding the first N elements

Single terminal operation

- **forEach(Consumer<T> action)** — Apply the provided action to each element of the stream
- **toArray()** — Create an array from the elements of the stream
- **reduce(...)** — Aggregate the elements of the stream into a summary value
- **collect(...)** — Aggregate the elements of the stream into a summary result container
- **min(Comparator<T>)** — Return the minimal element of the stream according to the comparator
- **max(Comparator<T>)** — Return the maximal element of the stream according to the comparator
- **count()** — Return the size of the stream
- **{any,all,none}Match(Predicate<T>)** — Return whether any/all/none of the elements of the stream match the provided predicate. **This is a short-circuiting operation**
- **findFirst()** — Return the first element of the stream, if present. **This is a short-circuiting operation**
- **findAny()** — Return any element of the stream, if present.

```java
Stream.of("1", "2", "3", "4", "5") // source   //sequential stream 1,2,3,4,5
.filter(s -> s.startsWith("1")) // intermediate operation
.map(String::toUpperCase) // intermediate operation
.sorted() // intermediate operation
.forEach(System.out::println); // terminal operation
```

```java
Stream.of("1", "2", "3", "4", "5")
.parallel() // parallel processing // 2,5,1,3,4
.filter(s -> s.startsWith("1"))
.map(String::toUpperCase)
.sorted()
.forEach(System.out::println);
```

```java
Arrays.asList("1", "2", "3", "4", "5")
.parallelStream() // parallel processing  // 2,5,1,3,4
.filter(s -> s.startsWith("1"))
.map(String::toUpperCase)
.sorted()
.forEach(System.out::println);
```

```java
Stream.of("1", "2", "3", "4","5")
.mapToInt(Integer::valueOf)
.sum();
// Output: 15
```

```java
Stream.of("R", "O", "B", "I", "N")
.collect(Collectors.joining());
// Output: ROBIN
```

```java
Stream.of("1", "2", "3", "4", "5")
.mapToInt(Integer::valueOf)
.reduce(0, (x,y) -> x+y);
// Output: 15
```

```java
Stream.of("R", "O", "B", "I", "N")
.reduce("", (a,b)->a+b);
// Output: ROBIN
```

Interface Extension Methods

Now default and static method is allowed in interface

Common areas in JDK where default methods are used:

- Several default methods like **replaceAll(), putIfAbsent(Key k, Value v) and others, added in the Map interface**
- **The forEach default method added to Iterable** interface
- **The Stream, parallelStream, spliterator methods added in Collection** interface

```
public interface Iterable<T> {
    Iterator<T> iterator();
    default void forEach(Consumer<? super T> action) {
        Objects.requireNonNull(action);
        for (T t : this) {
            action.accept(t);
        }
    }
    default Spliterator<T> spliterator() {
        return Spliterators.spliteratorUnknownSize(iterator(), 0);
    }
}
```

```
public interface Extension {
  default void printHello() {
    System.out.println("Hello from default method in interface");
  }
  static void printHello2() {
    System.out.println("Hello from static method in interface");
  }
}
public class ExtensionTest implements Extension {
  public static void main(String[] args) {
    ExtensionTest impl = new ExtensionTest();
    impl.printHello();
    Extension.printHello2();
  }
}
```

Optional:

Java introduced a new class in java 8. 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.

```
import java.util.Optional;
public class OptionalExample {
    public static void main(String[] args) {
        String[] str = new String[10];
        str[5] = "JAVA OPTIONAL CLASS EXAMPLE";  // Setting value for 5th index
        Optional<String> empty = Optional.empty();  // It returns an empty instance of Optional class
        System.out.println(empty);
        Optional<String> value = Optional.of(str[5]);  // It returns a non-empty Optional

        // It returns non-empty Optional if value is present, otherwise returns an empty Optional
        System.out.println("Nullable Optional: "+Optional.ofNullable(str[5]));
        // It returns value of an Optional. if value is not present, it throws an NoSuchElementException
        System.out.println("Getting value: "+value.get());
        System.out.println("Is value present: "+value.isPresent());        // It returns true if value is present, otherwise false

         // printing value by using method reference    [ public void ifPresent(Consumer<? super T> consumer) }]
        value.ifPresent(System.out::println);

      // If value is present, it returns an Optional otherwise returns an empty Optional
      //  [public Optional<T> filter(Predicate<? super T> predicate)]
        System.out.println("Filtered value: "+value.filter((s)->s.equals("Abc")));
        System.out.println("Filtered value: "+value.filter((s)->s.equals("JAVA OPTIONAL CLASS EXAMPLE")));

        // It returns value if available, otherwise returns specified value,
        System.out.println("orElse: "+value.orElse("Value is not present"));
        System.out.println("orElse: "+empty.orElse("Value is not present"));
        System.out.println("Getting hashCode: "+value.hashCode());  // It returns hashCode of the value
        System.out.println("Getting hashCode: "+value.toString());        // It returns string of the value

    }
}
```

Java 9 **new features**	1. Platform Module System (Project Jigsaw) 2. Interface Private Methods 3. Try-With Resources 4. Anonymous Classes 5. @SafeVarargs Annotation 6. Collection Factory Methods 7. Process API Improvement 8. New Version-String Scheme 9. JShell: The Java Shell (REPL) 10. Process API Improvement 11. Control Panel 12. Stream API Improvement 13. Installer Enhancement for Microsoft windows and many more

1. **Java Platform Module System** (Project Jigsaw) is a new kind of Java programing component that can be used to collect Java code (classes and packages). The main goal of this project is to easily scale down application to small devices. In Java 9, JDK itself has divided into set of modules to make it more lightweight.
2. **Interface allows us to declare private methods** that help to **share** common code between **non-abstract** methods.
3. Java introduced **try-with-resource** feature in Java 7 that helps to close resource automatically after being used. we don't need to close resources (file, connection, network etc) explicitly.

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

JAVA -7/8	**JAVA -9**

```
try(FileOutputStream fileStream=new FileOutputStrea
m("javatpoint.txt");){
            fileStream.write(b);
      }
catch(Exception e) {
         System.out.println(e);
 }
```
	```
	FileOutputStream fileStream=new FileOutputStre
	am("javatpoint.txt");

	 try(fileStream){
	            fileStream.write(b);
	      }
	catch(Exception e) {
	         System.out.println(e);
	      }
	```

4. **Anonymous classes improved (implemented diamond operator) :-** Using the diamond with anonymous classes was not allowed in Java 7.

```
abstract class ABCD<T>{
   abstract T show(T a, T b);
}
public class TypeInferExample {
   public static void main(String[] args) {
      ABCD<String> a = new ABCD<>() { // diamond operator is empty, compiler infer type
         String show(String a, String b) {
            return a+b;
         }
      };
      String result = a.show("Java","9");
      System.out.println(result);
   }
}
```

5. **@SafeVarargs Annotation** It is an annotation which applies on a method or constructor that takes **varargs parameters**. It is used to ensure that the method does not perform unsafe operations on its varargs parameters.

   It was included in Java7 and can only be applied on

- Final methods
- Static methods
- Constructors

From Java 9, it can also be used with **private instance methods**.
The @SafeVarargs annotation can be applied only to methods that cannot be overridden. Applying to the other methods will throw a compile time error.

```java
public class SafeVar{
    // Applying @SaveVarargs annotation
    @SafeVarargs
    private void display(List<String>... products) {
        for (List<String> product : products) {
            System.out.println(product);
        }
    }
    public static void main(String[] args) {
        SafeVar p = new SafeVar();
        List<String> list = new ArrayList<String>();
        list.add("Laptop");
        p.display(list);
    }
}
```

6. **Java Collection Factory Methods (of,ofEntries) :** Factory methods are special type of static methods that are used to create **unmodifiable instances** of collections. It means we can use these methods to create list, set and map of small number of elements. It is unmodifiable, so adding new element will throw **java.lang.UnsupportedOperationException.**

```java
List<String> list = List.of("Java","JavaFX","Spring","Hibernate","JSP");
        for(String l:list) {
            System.out.println(l);
        }
```

Map(of, ofEntries)

```java
Map<Integer,String> map = Map.of(101,"JavaFX",102,"Hibernate",103,"Spring MVC");
        for(Map.Entry<Integer, String> m : map.entrySet()){
            System.out.println(m.getKey()+" "+m.getValue());
        }

Map.Entry<Integer, String> e1 = Map.entry(101, "Java");
        Map.Entry<Integer, String> e2 = Map.entry(102, "Spring");
        Map<Integer, String> map = Map.ofEntries(e1,e2);
        for(Map.Entry<Integer, String> m : map.entrySet()){
            System.out.println(m.getKey()+" "+m.getValue());
        }
```

7. **Java Process API Improvement :-** Java has improved its process API in Java 9 version that helps to manage and control operating system processes.In earlier versions, it was complex to manage and control operating system processes by using Java programming. Now, new classes and interfaces are added to perform this task.New methods are added to the **java.lang.Process** class that are tabled below

Java ProcessHandle Interface: - ProcessHandle helps to handle and control processes. We can monitor processes, list its children, get information etc.This interface contains static factory methods that return instances that are value-based, immutable and thread-safe.

public interface ProcessHandle **extends** Comparable<ProcessHandle> : ProcessHandle helps to handle and control processes.
public static interface ProcessHandle.Info : used to provide information about the process.

EX:-

```java
public class ProcessApiExample {
    public static void main(String[] args) {
        ProcessHandle currentProcess = ProcessHandle.current();     // Current processhandle
        System.out.println("Process Id: "+currentProcess.pid());    // Process id
        System.out.println("Direct children: "+ currentProcess.children()); // Direct children of the process
        System.out.println("Class name: "+currentProcess.getClass());       // Class name
        System.out.println("All processes: "+ProcessHandle.allProcesses()); // All current processes
        System.out.println("Process info: "+currentProcess.info());         // Process info
        System.out.println("Is process alive: "+currentProcess.isAlive());
        System.out.println("Process's parent "+currentProcess.parent());  // Parent of the process
    }
}
```

8. **Java New Version-String Scheme**:- Java version-string is a format that contains version specific information. This version-string consists of major, minor, security and patch update releases.

Java Runtime.Version Example

```
public class VersionInfoExample {
   public static void main(String[] args) {
       Runtime.Version version = Runtime.version();              // Getting runtime version instance
       System.out.println("Current version is "+version);          // Getting current Java version
       System.out.println("Major version number "+version.major()); // Getting major version number
       System.out.println("Minor version number "+version.minor()); // Getting minor version number
       System.out.println("Security version number "+version.security());  // Getting security version number
       System.out.println("Pre-released information "+version.pre());      // Getting pre-
release version information
       System.out.println("Build Number "+version.build());              // Getting Optional build number
   }
}
```

9. **Java Shell Tool (JShell):** It is an interactive Java Shell tool, it allows us to execute Java code from the shell and shows output immediately. JShell is a REPL (Read Evaluate Print Loop) tool and run from the command line.

<u>Advantages of JShell</u>

Jshell has reduced all the efforts that are required to run a Java program and test a business logic.If we don't use Jshell, creating of Java program involves the following steps.

- Open editor and write program
- Save the program
- Compile the program
- Edit if any compile time error
- Run the program
- Edit if any runtime error
- Repeat the process

Jshell does not require above steps. We can evaluate statements, methods and classes, even can write hello program without creating class.

10. **<u>Java 9 Control Panel:</u>** Java control panel is used to control Java applications that are embedded in browser. This control panel maintains the settings that manage Java application embedded in browser.
11. **<u>Java 9 Stream API Improvement :</u>** In Java 9, Stream API has improved and new methods are added to the Stream interface. TakeWhile, dropWhile and ofNullable, and one overloaded iterate method are added to perform operations on stream elements.

takeWhile(): Stream takeWhile method takes each element that matches its predicate. It stops when it get unmatched element.

```
List<Integer> list  = Stream.of(1,2,3,4,5,6,7,8,9,10) .takeWhile(i -> (i % 2 == 0)).collect(Collectors.toList());
 System.out.println(list);  //[]
```

* This example returns an empty list because it fails at first list element, and takewhile stops here.

```
List<Integer> list  = Stream.of(2,2,3,4,5,6,7,8,9,10) .takeWhile(i -> (i % 2 == 0)).collect(Collectors.toList());
System.out.println(list);  //[2,2]
```

*we are getting first two elements because these are even and stops at third element.

dropWhile() : It returns a stream that contains elements after dropping the elements that match the given predicate.

```
List<Integer> list = Stream.of(2,2,3,4,5,6,7,8,9,10) .dropWhile(i -> (i % 2 == 0)).collect(Collectors.toList());
System.out.println(list);  // [3, 4, 5, 6, 7, 8, 9, 10]
```

ofNullable(): Stream ofNullable method returns a sequential stream that contains a single element, if non-null. Otherwise, it returns an empty stream.

```
Stream<Integer> val  = Stream.ofNullable(null);
val.forEach(System.out::println);
```

iterate(): A new overloaded method iterate is added to the Java 9 stream interface. This method allows us to iterate stream elements till the specified condition.

```
public static void main(String[] args) {
          Stream.iterate(1, i -> i <= 10, i -> i*3)
                .forEach(System.out::println);
}
```

Java-10 features:	**1 Java 10 Features** **1.1 Time-Based Release Versioning (JEP 322)** **1.2 Local-Variable Type Inference (JEP 286)** **1.3 Experimental Java-Based JIT Compiler (JEP 317)** **1.4 Application Class-Data Sharing (JEP 310)** **1.5 Parallel Full GC for G1 (JEP 307)** **1.6 Garbage-Collector Interface (JEP 304)** **1.7 Additional Unicode Language-Tag Extensions (JEP 314)** **1.8 Root Certificates (JEP 319)** **1.9 Thread-Local Handshakes (JEP 312)** **1.10 Heap Allocation on Alternative Memory Devices (JEP 316)** **1.11 Remove the Native-Header Generation Tool – javah (JEP 313)** **1.12 Consolidate the JDK Forest into a Single Repository (JEP 296)** **1.13 API Changes**

1 Java 10 Features

1.1 **Time-Based Release Versioning** Oracle changed the version-string scheme of the Java SE Platform and the JDK, and related versioning information, for present and future time-based release models.
$FEATURE.$INTERIM.$UPDATE.$PATCH

```
Version version = Runtime.version();
version.feature();
version.interim();
version.update();
version.patch();
```

1.2 **Local-Variable Type Inference**
Local-Variable Type Inference is the biggest new feature in Java 10 for developers. It adds type inference to declarations of local variables with initializers.

```
var numbers = List.of(1, 2, 3, 4, 5); // inferred value ArrayList<String>
// Index of Enhanced For Loop
for (var number : numbers) {
        System.out.println(number);
}
// Local variable declared in a loop
for (var i = 0; i < numbers.size(); i++) {
        System.out.println(numbers.get(i));
}
```

1.3 **Experimental Java-Based JIT Compiler** Graal was introduced in Java 9. It's an alternative to the JIT compiler which we have been used to. It's a plugin to the JVM, which means that the JIT compiler is not tied to JVM and it can be dynamically plugged in and replaced with any another plugin which JVMCI compliant (Java-Level JVM Compiler Interface).

1.4 Application Class-Data Sharing
JVM while starting performs some preliminary steps, one of which is loading classes in memory. If there are several jars having multiple classes, then the lag in the first request is clearly visible. This becomes an issue with serverless architecture, where the boot time is critical. In order to bring down application startup time, Application class-data sharing can be used. The idea is to reduce footprint by sharing common class metadata across different Java processes.

1.5 Parallel Full GC for G1
G1 garbage collector was made default in JDK 9. G1 Garbage collector avoids any full garbage collection, but when concurrent threads for collection cannot revive the memory fast enough users experience is impacted.

This change improves the G1 worst-case latency by making the full GC parallel. The mark-sweep-compact algorithm from G1 collector is parallelized as part of this change and will be triggered when concurrent threads for collection can't revive the memory fast enough.

1.6 Garbage-Collector Interface
It improves the code isolation of different garbage collectors by introducing a common Garbage Collector Interface.

This change provides better modularity to the Internal GC Code. It will help in the future for adding new GC without changing existing codebase, also help in removing

1.7 Additional Unicode Language-Tag Extensions
This feature enhances java.util.Locale and related APIs to implement additional Unicode extensions
1.cu (currency type)   2. fw (first day of week) 3. rg (region override) 4. tz (time zone)

1.8 Root Certificates
In order to promote OpenJDK and make it more appealing to community users, this feature provides a default set of root Certification Authority (CA) certificates in the JDK.

1.9 Thread-Local Handshakes

This is an internal JVM feature to improve performance.
A handshake operation is a callback that is executed for each JavaThread while that thread is in a safepoint state. The callback is executed either by the thread itself or by the VM thread while keeping the thread in a blocked state.

1.10 Heap Allocation on Alternative Memory Devices

This feature enhances the capability of HotSpot VM to allocate the Java object heap on an alternative memory device, such as an NV-DIMM, specified by the user.

1.11 Remove the Native-Header Generation Tool – javah

This is a housekeeping change to remove javah tool from JDK. The tool functionality is added in javac as part of JDK 8, which provides ability to write native header files at the compile-time rendering javah useless.

1.12 Consolidate the JDK Forest into a Single Repository

1.13 API Changes

Let's go through a few additions:

1. List, Map & Set Interfaces are added with a static copyOf(Collection) method. Its returns an unmodifiable List, Map or Set containing the entries provided. For a List, if the given List is subsequently modified, the returned List will not reflect such modifications.
2. Optional & its primitive variations get a method orElseThrow(). This is exactly same as get(), however the java doc states that it is a preferred alternative then get()
3. Collectors class gets various methods for collecting unmodifiable collections (Set, List, Map)

```
List<String> actors = new ArrayList<>();
actors.add("Jack Nicholson");
actors.add("Marlon Brando");
System.out.println(actors); // prints [Jack Nicholson, Marlon Brando]
// New API added - Creates an UnModifiable List from a List.
List<String> copyOfActors = List.copyOf(actors);
System.out.println(copyOfActors); // prints [Jack Nicholson, Marlon Brando]
// copyOfActors.add("Robert De Niro"); //Will generate an UnsupportedOperationException
actors.add("Robert De Niro");
System.out.println(actors);// prints [Jack Nicholson, Marlon Brando, Robert De Niro]
System.out.println(copyOfActors); // prints [Jack Nicholson, Marlon Brando]

String str = "";
Optional<String> name = Optional.ofNullable(str);
// New API added - is preferred option then get() method
name.orElseThrow(); // same as name.get()
```

Java-11 features:	**1. Running Java File with single command** **2. New utility methods in String class** **3. Local-Variable Syntax for Lambda Parameters** **4. Nested Based Access Control** **5. JEP 321: HTTP Client** **6. Reading/Writing Strings to and from the Files**

1. **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.

2**. New utility methods in String class**

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

```
System.out.println(" ".isBlank()); //true

        String s = "Anupam";
        System.out.println(s.isBlank()); //false
        String s1 = "";
        System.out.println(s1.isBlank()); //true
```

lines(): This method returns a stream of strings, which is a collection of all substrings split by lines.

```
String str = "JD\nJD\nJD";
        System.out.println(str);
        System.out.println(str.lines().collect(Collectors.toList()));
```

strip(), stripLeading(), stripTrailing()

strip() – Removes the white space from both, beginning and the end of string
But we already have trim(). Then what's the need of strip()?
strip() is "Unicode-aware" evolution of trim().
When trim() was introduced, Unicode wasn't evolved. Now, the new strip() removes all kinds of whitespaces leading and trailing(check the method Character.isWhitespace(c) to know if a unicode is whitespace or not)

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

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

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

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.

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

3. **Local-Variable Syntax for Lambda Parameters**

Local-Variable Syntax for Lambda Parameters is the only language feature release in Java 11.

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

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.
```

4. **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().

5. **JEP 321: HTTP Client**
   The new API supports both HTTP/1.1 and HTTP/2. It is designed to improve the overall performance of sending requests by a client and receiving responses from the server. It also natively supports WebSockets.

6. **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
```

Java-12 features:	1. **JVM Changes – JEP 189, JEP 346, JEP 344, and JEP 230.** 2. **Switch Expressions** 3. **File mismatch() Method** 4. **Compact Number Formatting** 5. **Teeing Collectors in Stream API** 6. **Java Strings New Methods – indent(), transform(), describeConstable(), and resolveConstantDesc().** 7. **JEP 334: JVM Constants API** 8. **JEP 305: Pattern Matching for instanceof** 9. **Raw String Literals is Removed From JDK 12.**

1. **JVM Changes**

A Low-Pause-Time Garbage Collector: RedHat initiated Shenandoah Garbage Collector to reduce GC pause times. The idea is to run GC concurrently with the running Java threads.
It aims at consistent and predictable short pauses irrelevant of the heap size. So it does not matter if the heap size is 15 MB or 15GB.
Promptly Return Unused Committed Memory from G1
Stating Java 12, G1 will now check Java Heap memory during inactivity of application and return it to the operating system. This is a preemptive measure to conserve and use free memory.
Abortable Mixed Collections for G1
Improvements in G1 efficiency include making G1 mixed collections abortable if they might exceed the defined pause target. This is done by splitting the mixed collection set into mandatory and optional.
Thus the G1 collector can prioritize on collecting the mandatory set first to meet the pause time goal.
Default CDS Archives
This enhances the JDK build process to generate a class data-sharing (CDS) archive, using the default class list, on 64-bit platforms. The goal is to improve startup time. From Java 12, CDS is by default ON.

To run your program with CDS turned off do the following:
```
java -Xshare:off HelloWorld.java
```
Now, this would delay the startup time of the program.

2. **Switch Expressions**

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.

1	2

```
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);
```
	```
	String result = "";
	        switch (day) {
	            case "M":
	            case "W":
	            case "F": {
	                result = "MWF";
	                break;
	            }
	            case "T":
	            case "TH":
	            case "S": {
	                result = "TTS";
	                break;
	            }
	        };
	System.out.println(result);
	```

1. With the new Switch expression, we don’t need to set break everywhere thus prevent logic errors!
2. run the below program containing the new Switch Expression using JDK 12.

```
public class SwitchExpressions {

    public static void main(String[] args)
    {
        System.out.println("New Switch Expression result:");
        executeNewSwitchExpression("M");
        executeNewSwitchExpression("TH");
        executeNewSwitchExpression("");
        executeNewSwitchExpression("SUN");
    }

    public static void executeNewSwitchExpression(String day){
        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);
    }
}
```

// Since this is a preview feature, please ensure that you have selected the Language Level as Java 12 preview.

3. **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();
    }
}
```

4. **Compact Number Formatting**

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

public class CompactNumberFormatting {
            public static void main(String[] args)
            {
                System.out.println("Compact Formatting is:");
                NumberFormat upvotes = NumberFormat
                        .getCompactNumberInstance(new Locale("en", "US"), NumberFormat.Style.SHORT);
                upvotes.setMaximumFractionDigits(1);
                System.out.println(upvotes.format(2592) + " upvotes");  // 2.6k upvotes

                NumberFormat upvotes2 = NumberFormat
                        .getCompactNumberInstance(new Locale("en", "US"), NumberFormat.Style.LONG);
                upvotes2.setMaximumFractionDigits(2);
                System.out.println(upvotes2.format(2011) + " upvotes"); // 2.01 thousand upvotes
            }
}
```

5. **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); // output 3.0
```

6. **Java Strings New Methods – indent(), transform(), describeConstable(), and resolveConstantDesc().**
7.

indent(int n)
This method adjusts the indentation of each line in the string based on the value of 'n' and also normalizes line termination characters.

If n > 0, then n spaces (U+0020) are inserted at the beginning of each line.
If n < 0, then up to n white space characters are removed from the beginning of each line. If a given line does not contain sufficient white space then all leading white space characters are removed. The tab character is also treated as a single character.
If n = 0, then the line remains unchanged. However, line terminators are still normalized.

```
String str = "*****\n  Hi\n  \tHello Pankaj\rHow are you?\n*****";
System.out.println(str.indent(0)); // Hi
System.out.println(str.indent(3)); // Hello Pankaj
System.out.println(str.indent(-3));// How are you?
```

transform(Function f)

This method allows us to call a function on the given string. The function should expect a single String argument and produce an R result.

```
String s = "Hi,Hello,Howdy";
List strList = s.transform(s1 -> {return Arrays.asList(s1.split(","));});
System.out.println(strList); // [Hi,Hello,Howdy]
```

Optional describeConstable()

string implements two new interfaces from Constants API – Constable, and ConstantDesc.
This method is declared in the Constable interface and implemented in the String class.

```
String so = "Hello";
Optional os = so.describeConstable();
System.out.println(os);  // [Hello]
System.out.println(os.get()); // Hello
```

String resolveConstantDesc(MethodHandles.Lookup lookup)
This method is part of Constants API and declared in ConstantDesc interface.
It resolves this instance as a ConstantDesc, the result of which is the instance itself.

```
String so1 = "Hello";
so1.resolveConstantDesc(MethodHandles.lookup()); // Hello
```

8. **JVM Constants API**
   A new package java.lang.constant is introduced with this JEP. This is not that useful for those developers who don't use constants pool.
9. **Pattern Matching for instanceof**

```
        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
}
```

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

String literal (removed)	**New way**
String html = "<html>\n" + " <body>\n" + " <p>Hello World.</p>\n" + " </body>\n" + "</html>\n";	String html = \`<html> <body> <p>Hello World.</p> </body> </html> \`;

Java-13 features:	1. **Text Blocks – JEP 355** 2. **New Methods in String Class for Text Blocks** 3. **Switch Expressions Enhancements – JEP 354** 4. **Reimplement the Legacy Socket API – JEP 353** 5. **Dynamic CDS Archive – JEP 350** 6. **ZGC: Uncommit Unused Memory – JEP 351** 7. **FileSystems.newFileSystem() Method** 8. **DOM and SAX Factories with Namespace Support**

1. **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.

```
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);
textBlock.equals(str) //true
```

2. **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.

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

System.out.println(output);
System.out.println(output.stripIndent().replace(" ", "*"));
String str1 = "Hi\t\nHello' \" /u0022 Pankaj\r";
System.out.println(str1); //// Hello' /u0022 Pankaj
System.out.println(str1.translateEscapes()); // Hello' /u0022 Pankaj
```

3. **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.

// upto java 12	// java 13 onwards - multi-label case statements	// java 13 switch expressions, uses yield to return, in Java 12 it was break

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");
 	
}
	switch (choice) {
	case 1, 2, 3:
	System.out.println(choice);
	break;
	default:
	System.out.println("integer is greater than 3");
	
	}
		int x = switch (choice) {
		case 1, 2, 3:
		yield choice;
		default:
		yield -1;
		};
		System.out.println("x = " + x);

4. **Reimplement the Legacy Socket API – JEP 353**
The underlying implementation of the java.net.Socket and java.net.ServerSocket APIs have been rewritten. The new implementation, NioSocketImpl, is a drop-in replacement for PlainSocketImpl.
It uses java.util.concurrent locks rather than synchronized methods.

5. **Dynamic CDS Archive – JEP 350**
This JEP extends the class-data sharing feature, which was introduced in Java 10. Now, the creation of CDS archive and using it is much easier.

```
$ java -XX:ArchiveClassesAtExit=my_app_cds.jsa -cp my_app.jar
$ java -XX:SharedArchiveFile=my_app_cds.jsa -cp my_app.jar
```

6. **ZGC: Uncommit Unused Memory – JEP 351**
This JEP has enhanced ZGC to return unused heap memory to the operating system. The Z Garbage Collector was introduced in Java 11. It adds a short pause time before the heap memory cleanup. But, the unused memory was not being returned to the operating system. This was a concern for devices with small memory footprint such as IoT and microchips. Now, it has been enhanced to return the unused memory to the operating system.

7. **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.
1.newFileSystem(Path)
2.newFileSystem(Path, Map<String, ?>)
3.newFileSystem(Path, Map<String, ?>, ClassLoader)

8. **DOM and SAX Factories with Namespace Support**

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

1.newDefaultNSInstance()
2.newNSInstance()
3.newNSInstance(String factoryClassName, ClassLoader classLoader)

```
//java 13 onwards
DocumentBuilder db = DocumentBuilderFactory.newDefaultNSInstance().newDocumentBuilder();

// before java 13
DocumentBuilderFactory dbf = DocumentBuilderFactory.newDefaultInstance();
dbf.setNamespaceAware(true);
DocumentBuilder db = dbf.newDocumentBuilder();
```

Java-14 features:	1. **Switch Expressions (Standard) – JEP 361** 2. **Pattern Matching for instanceof (Preview) – JEP 305** 3. **Helpful NullPointerExceptions – JEP 358** 4. **Records (Preview) – JEP 359** 5. **Text Blocks (Second Preview) – JEP 368**

1. **Switch Expressions (Standard) – JEP 361**
   Java 14 has finally made these features a standard now(->).

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

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

2. **Pattern Matching for instanceof (Preview) – JEP 305**
   Java 14, gets rid of this verbosity by making conditional extraction a lot more concise(obj type).

Before Java 14:	Java 14 Onwards:

if (obj instanceof Journaldev) {
  Journaldev jd = (Journaldev) obj;
  System.out.println(jd.getAuthor());
}
	if (obj instanceof Journaldev jd) {
	  System.out.println(jd.getAuthor());
	}

3. **Helpful NullPointerExceptions**
   It's an enhancement in the runtime environment.

4. **Records (Preview)**
   A record is a data class that stores pure data. The idea behind introducing records is to quickly create simple and concise classes devoid of boilerplate code.

   Normally a class in Java would require you to implement equals(), hashCode() , the getters and setters methods. While some IDEs support auto-generation of such classes, the code is still verbose. With a record you need to simply define a class in the following way.

```
record Author(){}
//or
record Author (String name, String topic) {}
```

   The Java compiler will generate a constructor, private final fields, accessors, equals/hashCode and toString methods automatically. The auto-generated getter methods of the above class are name() and topic().

```
record Author (int id, String name, String topic) {
    static int followers;

    public static String followerCount() {
        return "Followers are "+ followers;
    }

    public String description(){
        return "Author "+ name + " writes on "+ topic;
    }

    public Author{
    if (id < 0) {
        throw new IllegalArgumentException( "id must be greater than 0.");
     }
    }
}
```

The additional constructor defined inside the record is called a Compact constructor. It doesn't consist of any parameters and is just an extension of the canonical constructor.

A compact constructor wouldn't be generated as a separate constructor by the compiler. Instead, it is used for validation cases and would be invoked at the start of the main constructor.

Few important things to note about Records:
**A record can neither extend a class nor it can be extended by another class. It's a final class.*
**Records cannot be abstract*
**Records cannot extend any other class and cannot define instance fields inside the body. Instance fields must be defined in the state description only*
**Declared fields are private and final*
**The body of a record allows static fields and methods*

1.Records Can Implement Interfaces
The following code shows an example of implementing an interface with records:

```
record Author(String name, String topic) implements Information {
  public String getFullName() {
    return "Author "+ name + " writes on " + topic;
  }
}

interface Information {
  String getFullName();
}
```

2.Records support multiple constructors

```
record Author(String name, String topic) {
  public Author() {

    this("NA", "NA");
  }

  public Author(String name) {

    this(name, "NA");
  }
}
```

2.Records Allow Modifying Accessor Methods

Though records do generate public accessor methods for the fields defined in the state description, they also allow you to redefine the accessor methods in the body as shown below:

```
record Author(String name, String topic) {
  public String name() {
        return "This article was written by " + this.name;
    }
}
```

5. Text Blocks (Second Preview)

In Java 14, Text Blocks are still in preview with some new additions. We can now use:

Backslash for displaying nice-looking multiline string blocks.
\s is used to consider trailing spaces which are by default ignored by the compiler. It preserves all the spaces present before it.

```
String text2 = """
                line1
                line2 \s    //space before \s if preserved
                line3
                """;
```