Java 9 Features

1. Java platform module system

- A module is new construct like we already have packages. An application, developed using new modular programming, can be seen as collection of interacting modules with a well-defined boundaries and dependencies between those modules.
- The JPMS consists of providing support for writing modular applications as well as modularizing the JDK source code as well.
- JDK 9 is coming with around 92 modules (changes are possible in GA release). Java 9 Module System has a "java.base" Module. It's known as Base Module.
- It's an Independent module and does NOT dependent on any other modules. By default, all other modules are dependent on "java.base".
- A module is typically just a jar file that has a `module-info.class` file at the root. To use a module, include the jar file into `module path` instead of the `classpath`.
- A modular jar file added to classpath is normal jar file and `module-info.class` file will be ignored.

---

2. Interface Private Methods

- Java 9 onward, you are allowed to include private methods in interfaces.
- These private methods will improve code re-usability inside interfaces.
- For example, if two default methods needed to share code, a private interface method would allow them to do so, but without exposing that private method to it's implementing classes.
- Using private methods in interfaces have four rules :
    - Private interface method cannot be abstract.

- Private method can be used only inside interface.
- Private static method can be used inside other static and non-static interface methods.
- Private non-static methods cannot be used inside private static methods.

---

3. HTTP/2 Client

- HTTP/1.1 client was released on 1997. A lot has changed since. So for Java 9 a new API been introduced that is cleaner and clearer to use and which also adds support for HTTP/2.
- New API uses 3 major classes i.e. HttpClient, HttpRequest and HttpResponse.

```
HttpClient httpClient = HttpClient.newHttpClient();
HttpRequest httpRequest = HttpRequest.newBuilder().uri(new URI("//
howtodoinjava.com/")).GET().build();
HttpResponse<String> httpResponse = httpClient.send(httpRequest,
HttpResponse.BodyHandler.asString());
System.out.println( httpResponse.body() );
```

- New API also support Async HTTP requests using httpClient.sendAsync() method.
- It returns CompletableFuture object which can be used to determine whether the request has been completed or not.
- It also provide you access to the HttpResponse once request is completed. Best part is that if you desire you can even cancel the request before it completes.

---

4. JShell – REPL Tool

- JShell is new command line interactive tool shipped with JDK 9 distribution to evaluate declarations, statements and expressions written in Java.
- JShell allows us to execute Java code snippets and get immediate results without having to create a solution or project.

- Jshell is much like what we have command window in linux OS. Difference is that JShell is Java specific. It has lots of other capabilities, other than executing simple code snippets. e.g.
    - Launch inbuilt code editor in separate window
    - Launch code editor of your choice in separate window
    - Execute code when Save operation happen in these external editors
    - Load pre-written classes from file system

---

5. Platform and JVM Logging

- JDK 9 has improved logging in platform classes (JDK classes) and JVM components, through a new loging API. It lets you specify a logging framework of your choice (e.g. Log4J2) as logging backend for logging messages from JDK classes.

---

6. Process API Updates

- Prior to Java 5, the only way to spawn a new process was to use the Runtime.getRuntime().exec() method. Then in Java 5, ProcessBuilder API was introduced which supported a cleaner way of spawning new processes. Now Java 9 is adding a new way of getting information about current and any spawned process.
- To get information of any process, now you should use java.lang.ProcessHandle.Info interface. This interface can be useful in getting lots of information e.g.
    - the command used to start the process
    - the arguments of the command
    - time instant when the process was started
    - total time spent by it and the user who created it

7. Collection API Updates

- Since Java 9, you can create immutable collections such as immutable list, immutable set and immutable map using new factory methods. e.g.

```
public class ImmutableCollections
{
   public static void main(String[] args)
   {
      List<String> namesList = List.of("Lokesh", "Amit", "John");

      Set<String> namesSet = Set.of("Lokesh", "Amit", "John");

      Map<String, String> namesMap = Map.ofEntries(
                        Map.entry("1", "Lokesh"),
                        Map.entry("2", "Amit"),
                        Map.entry("3", "Brian"));
   }
}
```

8. Stream API Improvements

- Java 9 has introduced two new methods to interact with Streams
  i.e. takeWhile / dropWhile methods.
- Additionally, it has added two overloaded methods i.e. ofNullable and iterate methods

Java 10 Features

1. Local Variable Type Inference
    1. It allows you to declare a local variable without specifying its type. The type of variable will be inferred from type of actual object created.

```
var str = "Hello world";

//or

String str = "Hello world";
```

2. Time-Based Release Versioning

    $FEATURE.$INTERIM.$UPDATE.$PATCH

3. Garbage-Collector Interface
    - In earlier JDK structure, the components that made up a Garbage Collector (GC) implementation were scattered throughout various parts of the code base. It's changed in Java 10. Now, it is a clean interface within the JVM source code to allow alternative collectors to be quickly and easily integrated. It will improve source-code isolation of different garbage collectors.
4. Parallel Full GC for G1
    - The current implementation of the full GC for G1 uses a single threaded mark-sweep-compact algorithm. This change will parallelize the mark-sweep-compact algorithm and use the same number of threads. It will be triggered when concurrent threads for collection can't revive the memory fast enough.
5. Heap Allocation on Alternative Memory Devices
6. Consolidate the JDK Forest into a Single Repository

7. Application Class-Data Sharing
8. Additional Unicode Language-Tag Extensions
9. Root Certificates
10. Experimental Java-Based JIT Compiler
11. Thread-Local Handshakes
12. Remove the Native-Header Generation Tool
13. New Added APIs and Options
14. Removed APIs and Options

Java 11 Features

1. Launch Single-File Programs Without Compilation
    - Traditionally, for every program that we'd like to execute, we need to first compile it. It seems unnecessarily lengthy process for small programs for testing purposes.
    - Java 11 changes it and now we can execute Java source code contained in a single file without the need to compile it first.
2. String API Changes

    1. String.repeat(Integer)
        This method simply repeats a string n times. It returns a string whose value is the concatenation of given string repeated N times.

        ```
        public class HelloWorld
        {
           public static void main(String[] args)
           {
              String str = "1".repeat(5);

              System.out.println(str);    //11111
           }
        }
        ```

    2. String.isBlank()
        This method indicates whether a string is empty or contains only white-spaces. Previously, we have been using it from Apache's StringUtils.java.

        ```
        public class HelloWorld
        {
           public static void main(String[] args)
           {
              "1".isBlank();  //false

              "".isBlank();   //true
        ```

```
        "    ".isBlank();   //true
    }
}
```

3. String.strip()

    This method takes care of removing leading and trailing white-spaces. We can be even more specific by removing just the leading characters by using **String.stripLeading()** or just the trailing characters by using **String.stripTrailing()**.

```
public class HelloWorld
{
    public static void main(String[] args)
    {
        "   hi  ".strip();  //"hi"

       "   hi  ".stripLeading();    //"hi   "

       "   hi  ".stripTrailing();   //"   hi"
    }
}
```

4. String.lines()

    This method helps in processing multi-line texts as a **Stream**.

```
public class HelloWorld
{
    public static void main(String[] args)
    {
        String testString = "hello\nworld\nis\nexecuted";

        List<String> lines = new ArrayList<>();

        testString.lines().forEach(line -> lines.add(line));

        assertEquals(List.of("hello", "world", "is", "executed"), lines);
    }
}
```

3. Collection.toArray(IntFunction)

Before Java 11, converting a collection to array was not straightforward. Java 11 makes the conversion more convenient.

```
public class HelloWorld
{
   public static void main(String[] args)
   {
      List<String> names = new ArrayList<>();
      names.add("alex");
      names.add("brian");
      names.add("charles");

      String[] namesArr1 = names.toArray(new String[names.size()]);       //Before Java 11

      String[] namesArr2 = names.toArray(String[]::new);                  //Since Java 11
   }
}
```

4. Files.readString() and Files.writeString()

Using these overloaded methods, Java 11 aims to reduce a lot of boilerplate code which makes much easier to read and write files.

```
public class HelloWorld
{
   public static void main(String[] args)
   {
      //Read file as string
      URI txtFileUri =
getClass().getClassLoader().getResource("helloworld.txt").toURI();

      String content =
Files.readString(Path.of(txtFileUri),Charset.defaultCharset());

      //Write string to file
      Path tmpFilePath = Path.of(File.createTempFile("tempFile",
".tmp").toURI());

      Path returnedFilePath = Files.writeString(tmpFilePath,"Hello World!",
                              Charset.defaultCharset(),
StandardOpenOption.WRITE);
   }
}
```

5. Optional.isEmpty()

Optional is a container object which may or may not contain a non-null value. If no value is present, the object is considered empty.
Previously existing method **isPresent()** returns true if a value is present, otherwise false. Sometimes, it forces us to write negative conditions which are not readable.

isEmpty() method is reverse of isPresent() method and returns false if a value is present, otherwise true.
So we do not to write negative conditions in any case. Use any of these two methods when appropriate.

```
public class HelloWorld
{
    public static void main(String[] args)
    {
        String currentTime = null;

        assertTrue(!Optional.ofNullable(currentTime).isPresent());  //It's
negative condition
        assertTrue(Optional.ofNullable(currentTime).isEmpty());     //Write it
like this

        currentTime = "12:00 PM";

        assertFalse(!Optional.ofNullable(currentTime).isPresent()); //It's
negative condition
        assertFalse(Optional.ofNullable(currentTime).isEmpty());    //Write it
like this
    }
}
```

Java 12 Features

1. Collectors.teeing() in Stream API

A teeing collector has been exposed as a static method **Collectors::teeing**. This collector forwards its input to two other collectors before merging their results with a function.
teeing(Collector, Collector, BiFunction) accepts two collectors and a function to merge their results. Every element passed to the resulting collector is processed by both downstream collectors, then their results are merged using the specified merge function into the final result.
For example, in a given list of employees, if we want to find out the employee with maximum salary and minimum salary, we can do it in single statement using teeing collector.
Collector.teeing() Example

```
SalaryRange salaryRange = Stream
        .of(56700, 67600, 45200, 120000, 77600, 85000)
        .collect(teeing(
                minBy(Integer::compareTo),
                maxBy(Integer::compareTo),
                SalaryRange::fromOptional));
```

Read More : Collectors.teeing()

2. String API Changes

2.1. String.indent()

The indent method helps with changing the indentation of a String. We can either pass a positive value or a negative value depending on whether we want to add more white spaces or remove existing white spaces.

```
String result = "foo\nbar\nbar2".indent(4);

System.out.println(result);

//    foo
//    bar
//    bar2
```

Please note that the indent() method automatically adds a newline character if it hasn't been provided yet. That's to be expected and is a feature of the new method.
Each white space character is treated as a single character. In particular, the tab character "\t" (U+0009) is considered a single character; it is not expanded.

2.2. String.transform()

The transform() method takes a String and transforms it into a new String with the help of a Function.
In given example, we have a list of names. We are performing two operations (trimming white spaces and making all names camel-case)
using transform() method.

```
List<String> names = List.of(
                    "   Alex",
                    "brian");

List<String> transformedNames = new ArrayList<>();

for (String name : names)
{
   String transformedName = name.transform(String::strip)
                           .transform(StringUtils::toCamelCase);

   transformedNames.add(transformedName);
}
```

2.3. String constants

Since Java 12, String class implements two additional interfaces **java.lang.constant.Constable** and **java.lang.constant.ConstantDesc**.
String class also introduces two additional low-level methods describeConstable() and resolveConstantDesc(MethodHandles.Lookup).
They are low-level APIs meant for libraries and tools providing bytecode parsing and generation functionality, for example, Byte Buddy.
Just to note, a Constable type is one whose values are constants that can be represented in the constant pool of a Java class file as described in JVMS 4.4, and whose instances can describe themselves nominally as a ConstantDesc.
resolveConstantDesc() is similar to describeConstable() with the difference being that this method returns an instance of ConstantDesc instead.

3. Files.mismatch(Path, Path)

Sometimes, we want to determine whether two files have the same content. This API helps in comparing the content of files.
mismatch() method compares two file paths and return a long value. The long indicates the **position of the first mismatched byte** in the content of the two files. The return value will be '–1' if the files are “equal.”

```
Path helloworld1 = tempDir.resolve("helloworld1.txt");

Path helloworld2 = tempDir.resolve("helloworld2.txt");

long diff = Files.mismatch(helloworld1, helloworld2);   //returns long value
```

4. Compact Number Formatting

Large numbers rendered by a user interface or a command-line tool are always difficult to parse. It is far more common to use the abbreviated form of a number. Compact number representations are much easier to read and require less space on the screen without losing the original meaning.

E.g. **3.6 M** is very much easier to read than **3,600,000**.

Java 12 introduces a convenient method called **NumberFormat.getCompactNumberInstance(Locale, NumberFormat.Style)** for creating a compact number representation.

```
NumberFormat formatter =
NumberFormat.getCompactNumberInstance(Locale.US,
                                      NumberFormat.Style.SHORT);
```

5. Support for Unicode 11

In a time in which emojis play a crucial role in communicating on social media channels, it's more important than ever to support the latest Unicode specification. Java 12 has kept pace and supports Unicode 11.

Unicode 11 adds 684 characters, for a total of 137,374 characters – and seven new scripts, for a total of 146 scripts.

6. Switch Expressions (Preview)

This change extends the **switch statement** so that it can be used as either a statement or an expression.

Instead of having to define a break statement per case block, we can simply use the **arrow syntax**. The arrow syntax semantically looks like a lambda and separates the case label from the expression.

With the new switch expressions, we can directly assign the switch statement to a variable.

```
boolean isWeekend = switch (day)
{
    case MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY -> false;

    case SATURDAY, SUNDAY -> true;

    default -> throw new IllegalStateException("Illegal day entry :: " + day);
};

System.out.println(isWeekend);  //true or false - based on current day
```

To use this preview feature, remember that we have to explicitly instruct the JVM during application startup using **–enable-preview** flag.

Java 13 Features

1. JEP 350 Dynamic CDS Archives

- The Class Data Sharing (CDS) improves **startup performance** by creating a class-data archive once and reusing it so that the JVM needs not to recreate it again

2. JEP 351 ZGC: Uncommit Unused Memory

-This JEP enhanced the ZGC by returning unused heap memory to the operating system.

3. JEP-353 Reimplement the Legacy Socket API

- This JEP introduces new underlying implementations for the Socket APIs, which is the default implementation in Java 13.

- Before Java 13, it uses the `PlainSocketImpl` for the `SocketImpl`

4. JEP-354 Switch Expressions

- This JEP dropped `break value` in favor of `yield` keyword to return a value from `switch` expressions.

```
private static String getNumberViaYield(int number) {
      return switch (number) {
          case 1, 2:
              yield "one or two";
          case 3:
              yield "three";
          case 4, 5, 6:
              yield "four or five or six";
          default:
              yield "unknown";
      };
  }
```

5. JEP-355 Text Blocks

- This JEP introduces a multi-line string literal, a text block, finally.

Before Java 13

```
 String html ="<html>\n" +
              "   <body>\n" +
              "      <p>Hello, World</p>\n" +
              "   </body>\n" +
              "</html>\n";


 String json ="{\n" +
              "   \"name\":\"mkyong\",\n" +
              "   \"age\":38\n" +
              "}\n";
```

Now Java 13

```
 String html =  """
                <html>
                    <body>
                        <p>Hello, World</p>
                    </body>
                </html>
                        """;
```

```
String json = """
              {
                  "name":"mkyong",
                  "age":38
              }
              """;
```

Java 14 Features

1. JEP 305 – Pattern Matching for instanceof (Preview)

- In Java 14, instanceof operator has been modified to have **type test pattern**. A type test pattern (used in instanceof) consists of a predicate that specifies a type, along with a single binding variable.

```
if (obj instanceof String s) {
    // can use s here
} else {
    // can't use s here
}
```

2. **JEP 358 – Helpful NullPointerExceptions**

- Java 14 improves the usability of NullPointerException generated by the JVM by describing precisely which variable was null.
-

3. JEP 359 – Records (Preview)

- record type has been introduced as preview feature in Java 14 and shall be used as plain immutable data classes for data transfer between classes and applications.

Like enum, record is also a special class type in Java.

- It is intended to be used in places where a class is created only to act as plain data carrier.

```
public record EmployeeRecord(Long id,
        String firstName,
```

```
        String lastName,
        String email,
        int age) {

}
```

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

Java 15 Features

1. Sealed Classes and Interfaces

- Prior to Java 15, there was no restriction on classes or interfaces regarding which all classes can inherit them.
- A sealed class or interface restricts which other classes or interfaces may extend or implement them.

```
sealed class Account
    permits CurrentAccount, SavingAccount, LoanAccount {
}
```

2. EdDSA Algorithm

- EdDSA (Edwards-Curve Digital Signature Algorithm) is another additional digital signature scheme added in Java 15

3. Hidden Classes

Hidden classes are different from normal Java classes. They cannot be used directly by the bytecode of other classes. Hidden classes are intended for use by frameworks that generate classes at run time and use them indirectly, via reflection.

Java 16 Features

1. JEP 338: Vector API (Incubator)
2. JEP 347: Enable C++14 Language **Features**.
3. JEP 357: Migrate from Mercurial to Git.
4. JEP 369: Migrate to GitHub.
5. JEP 376: ZGC: Concurrent Thread-Stack Processing.
6. JEP 380: Unix-Domain Socket Channels.
7. JEP 386: Alpine Linux Port.
8. JEP 387: Elastic Metaspace.

Java 17 Features For Developers

1. Restore Always-Strict Floating-Point Semantics

This JEP is targeted toward scientific calculations that involve floating-point numbers. To guarantee the same calculation result on all platforms, we have been using the keyword strictfp.

The *strictfp* modifier accomplishes this by representing all intermediate values as IEEE single-precision and double-precision values. However, due to hardware heating issues, it became optional in JDK 1.2.

Today, as the hardwares has evolved and those heating issues are fixed, default floating-point semantics have been changed to *consistently strict*.

```
public strictfp double sum()
    {

        double num1 = 10e+10;
        double num2 = 6e+08;

        // Returning the sum
        return (num1 + num2);
```

```
    }
```

2. Pattern Matching for Switch (Preview)

Traditional switch statements throw NullPointerException if the selector expression evaluates to null. With this change, we can check for such *null* expressions as the separate *case* itself.

Before Java 17

```
if (s == null) {
   System.out.println("oops!");
   return;
}
switch (s) {
   case "Foo", "Bar" -> System.out.println("Great");
   default           -> System.out.println("Ok");
}
```

In Java 17

```
switch (s) {
   case null         -> System.out.println("Oops");
   case "Foo", "Bar" -> System.out.println("Great");
   default           -> System.out.println("Ok");
}
```

Improved instanceof checking

Before Java 16

```
Object o;
if (o instanceof String)
{
   String s = (String) o;
   String.format("String %s", s)
}
```

```
else if (o instanceof Integer)
{
    Integer i = (Integer) o;
    String.format("int %d", i)
}
else if (o instanceof Double)
{
    Double d = (Double) o;
    String.format("double %f", d)
}
```

In Java 16

```
Object o;
if (o instanceof String s)
{
    String.format("String %s", s)
}
else if (o instanceof Integer i)
{
    String.format("int %d", i)
}
else if (o instanceof Double d)
{
    String.format("double %f", d)
}
```

Java 17 takes it to the next level with switch expression. Now we can rewrite the above code as:

In Java 17

```
Object o;
switch (o)
{
    case Integer i -> String.format("int %d", i);
    case Double d  -> String.format("double %f", d);
    case String s  -> String.format("String %s", s);
    default        -> o.toString();
```

```
}
```

3.Sealed Classes

Sealed classes and interfaces restrict which other classes or interfaces may extend or implement them.

```
public sealed class Account
    permits CurrentAccount, SavingAccount, LoanAccount {
}
```

It was a preview feature in Java 15 and Java 16. In Java 17, it has become a standard feature with no changes to what was available in Java 16.

4. Enhanced Pseudo-Random Number Generators

```
RandomGeneratorFactory factory =
RandomGeneratorFactory.of("SecureRandom")
RandomGenerator random = factory.create(200L);

//get random numbers
randomGenerator.nextDouble();
```

5. Deprecate the Applet API for Removal

Most of the web browsers have already removed the support for Applets due to security concerns. In Java 9, Applet API was marked deprecated.

Since it has become irrelevant today, Java 17 has marked it for removal.

6. Strongly Encapsulate JDK Internals

This JEP strongly encapsulates all internal elements of the JDK, except for critical internal APIs such as *sun.misc.Unsafe*.

From Java 9 to Java 16, developers were able to access the JDK internal APIs using the flag –illegal-access. Java 17 will ignore the flag, and if the flag is present.

7. Remove RMI Activation

Remove the Remote Method Invocation (RMI) Activation mechanism while preserving the rest of RMI. It is obsolete and has been deprecated in Java 15.

8.Remove the Experimental AOT and JIT Compiler

This JEP removes the experimental Java-based ahead-of-time (AOT) and just-in-time (JIT) compiler introduced in Java 9 and Java 10.

The Graal compiler was made available as an experimental JIT compiler in JDK 10 via JEP 317. Since they were introduced, there was little use of these experimental features, and the effort required to maintain and enhance them was significant.

The developers can still leverage these features using GraalVM.

Java 21 Features

1. Virtual Threads

The virtual threads are JVM-managed lightweight threads that will help in writing high-throughput concurrent applications

With the introduction of virtual threads, it becomes possible to execute millions of virtual threads using only a few operating system threads. The most advantageous aspect is that there is no need to modify existing Java code. All that is required is instructing our application framework to utilize virtual threads in place of platform threads.

```
Runnable runnable = () -> System.out.println("Inside Runnable");

//1
Thread.startVirtualThread(runnable);

//2
Thread virtualThread = Thread.ofVirtual().start(runnable);

//3
var executor = Executors.newVirtualThreadPerTaskExecutor();
executor.submit(runnable);
```

2. Sequenced Collections

The new interfaces created under the sequenced collections initiative represent **collections with a defined encounter order**. The order will have a well-defined first element, second element, and so forth, up to the last element. The newly added interfaces provide a uniform API to access these elements in a sequence, or in the reverse order.

```
interface SequencedCollection<E> extends Collection<E> {
```

```
  // new method
  SequencedCollection<E> reversed();

  // methods promoted from Deque
  void addFirst(E);
  void addLast(E);
  E getFirst();
  E getLast();
  E removeFirst();
  E removeLast();
}
```

Record Patterns

Previously, if we need to access the components of a *record*, we should destructure it as follows:

```
Point obj = new Point(1,2);

if (obj instanceof Point p) {

  int x = p.x();
  int y = p.y();
  System.out.println(x+y);  //Prints 3
}
```

The record patterns eliminate the declaration of local variables for extracted components and initialize the components by invoking the accessor methods when a value is matched against the pattern.

```
if (obj instanceof Point(int x, int y)) {

    System.out.println(x+y);
}
```

Pattern Matching for *switch*

For example, in Java 16, we could have done something like this:

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

public void print(Object o) {

  switch (o) {

    case Point p                -> System.out.printf("o is a position: %d/%d%n", p.x(),
p.y());
    case String s   -> System.out.printf("o is a string: %s%n", s);
    default         -> System.out.printf("o is something else: %s%n", o);
  }
}
```

In Java 21, we can write the similar expression with *record* pattern as follows:

```
public void print(Object o) {

  switch (o) {

    case Point(int x, int y)                    -> System.out.printf("o is a position: %d/
%d%n", x, y);
    case String s               -> System.out.printf("o is a string: %s%n", s);
    default                     -> System.out.printf("o is something else: %s%n", o);
  }
}
```

String Templates (Preview)

Using the string templates, we can create string templates containing embedded expressions (evaluated at runtime). The template strings can contain variables, methods or fields, computed at run time, to produce a formatted string as output.

Syntactically, a template expression resembles a string literal with a prefix.

```
String message = STR."Greetings \{ name }!";  //Java
```

Unnamed Patterns and Variables (Preview)

Let us understand with an example. We typically handle the exception as follows:

```
String s = ...;

try {
    int i = Integer.parseInt(s);
    //use i
} catch (NumberFormatException ex) {
    System.out.println("Invalid number: " + s);
}
```

Notice we have created the variable ex but we used it nowhere. In the above example, the variable is unused and its name is irrelevant. The unnamed variables feature allows us to skip naming the variable and simply use an underscore (_) in place of it. Here **underscore signifies the absence of a name**.

```
String s = ...;

try {
```

```
    int i = Integer.parseInt(s);
    //use i
} catch (NumberFormatException _) {
    System.out.println("Invalid number: " + s);
}
```