**Java 8,9,10,11 Features – 2022**

**Interface Default and Static Methods**

* Before Java 8, interfaces could have only public abstract methods.
* It was not possible to add new functionality without forcing all implementing classes to create an implementation of the new methods.
* Starting with Java 8, interfaces can have ***static*** and **default** methods that.

**Static Method**

* Static method can't be overridden by an implementing class.

public interface MyJava8Interface {  
 static String getCountryCode(String country) {  
 return "ISO-Code";  
 }  
  
 static String getName() {  
 return "someName";  
 }  
}

Usage

String val1 = MyJava8Interface.*getCountryCode*("India");  
String val2 = MyJava8Interface.*getName*();  
System.*out*.println(val1+"<--->"+val2);

**Why Static Method?**

* To create a utility method.
* The idea behind static interface methods is to provide a simple mechanism that allows us to **increase the degree of cohesion** of a design by putting together related methods in one single place without having to create an object
* **To add new functionality in the interfaces without breaking the existing contract of the implementing classes.**
* The main difference is that **abstract classes can have constructors, state, and behavior**.

**Default Method**

Default methods are declared using the new **default keyword.** These are accessible through the instance of the implementing class and can be overridden.

**Why Interfaces Need Default Methods**

* **default methods are implicitly public;**there's no need to specify the public modifier.
* **backward compatibility is neatly preserved**

**How does it solve the diamond problem?**

public interface Author {  
 default void write() {  
 System.*out*.println("Author writing ...");  
 }  
}

public interface Writer {  
 default void write() {  
 System.*out*.println("Writer writing ...");  
 }  
}

**You can’t write like this, as it creates ambiguities, it gives compilation issue.**

public class Novelist implements Author, Writer {  
 public static void main(String[] args) {  
 Novelist novelist = new Novelist();  
 novelist.write();  
 }  
}

**You can Write the Novelist class as**

public class Novelist implements Author, Writer {  
 @Override  
 public void write() {  
 **Author.super.write(); // Author writing ...** }  
  
}

Or you can completely override the method.

public class Novelist implements Author, Writer {  
 @Override  
 public void write() {  
 System.*out*.println("Novelist writing");  
 }  
}

In this case, **the code simply won't compile, as there's a conflict caused by multiple interface inheritance** (a.k.a the [Diamond Problem](https://en.wikipedia.org/wiki/Multiple_inheritance)).

**To solve this ambiguity, we must explicitly provide an implementation for the methods:**

**Method References (Double Colon ::)**

The reference to a static method holds the syntax **ContainingClass::methodName.**

**boolean** isReal = list.stream().anyMatch(u -> User.isRealUser(u));

can be written as

**boolean isReal = list.stream().anyMatch(User::isRealUser);**

**Reference to a Constructor**

A reference to a constructor takes the syntax**ClassName::new.**

**Optional<T>**

Java 8 Optional<T> class can help to handle situations when there is a possibility of **NullPointerException**.

**Functional Interfaces and Lambda Expressions**

**Java Stream API for Bulk Data Operations on Collections**

**Java IO improvements**

Path filePath = Paths.get("c:/temp", "data.txt");

List<String> lines = Files.readAllLines(filePath);

Stream<String> lines = Files.lines( filePath ) 🡸 Reading File

String text = "Text to save to file";

Files.write(Paths.get("./fileName.txt"), text.getBytes()); 🡸 Writing file

**New Features in Java 9**

**Modular System – Jigsaw Project**

**A New HTTP Client**

HttpRequest request = HttpRequest.newBuilder()

.uri(new URI("https://postman-echo.com/get")) .GET() .build();

HttpResponse<String> response = HttpClient.newHttpClient() .send(request, HttpResponse.BodyHandler.asString());

**Process API**

The process API has been improved for controlling and managing operating-system processes.

ProcessHandle self = ProcessHandle.*current*();  
long PID = self.pid();  
ProcessHandle.Info procInfo = self.info();  
Optional<String[]> args = procInfo.arguments();  
Optional<String> cmd = procInfo.commandLine();  
Optional<Instant> startTime = procInfo.startInstant();  
Optional<Duration> cpuUsage = procInfo.totalCpuDuration();

**Destroying Processes**

Now – let's stop all the running child processes using **destroy()**

childProc = ProcessHandle.current().children();

childProc.forEach(procHandle -> {

assertTrue("Could not kill process " + procHandle.getPid(), procHandle.destroy());

});

**Interface Private Method**

Interfaces in the upcoming JVM version can have private methods, which can be used to split lengthy default methods

public interface Foo {

default void bar() {

System.out.print("Hello");

baz();

}

private void baz() {

System.out.println(" world!");

}

}

**JShell Command Line Tool**

**JCMD Sub-Commands**

We will get a list of all classes loaded in the JVM and their inheritance structure.

In the example below we can see the hierarchy of java.lang.Socket loaded in JVM running Eclipse Neon:

jdk-9\bin>jcmd 14056 VM.class\_hierarchy -i -s java.net.Socket

14056:

java.lang.Object/null

|--java.net.Socket/null

| **implements** **java**.io.Closeable/null (declared intf)

| **implements** **java**.lang.AutoCloseable/null (inherited intf)

**Variable Handles**

The API resides under java.lang.invoke and consists of VarHandle and MethodHandles. It provides equivalents of java.util.concurrent.atomic and sun.misc.Unsafe operations upon object fields and array elements with similar performance.

**With Java 9 Modular system access to sun.misc.Unsafe will not be possible from application code.**

**Publish-Subscribe Framework**

The class java.util.concurrent.Flow provides interfaces that support the [Reactive Streams](http://www.reactive-streams.org/) publish-subscribe framework. These interfaces support interoperability across a number of asynchronous systems running on JVMs. We can use utility class SubmissionPublisher to create custom components.

**New Features in Java 10**

**copyOf()**

java.util.List, java.util.Map and java.util.Set each got a new static method copyOf(Collection).

It returns the unmodifiable copy of the given Collection:

List<Integer> copyList = **List.copyOf(someIntList)**;

copyList.add(4);

**toUnmodifiable\*()**

java.util.stream.Collectors get additional methods to collect a Stream into unmodifiable List, Map or Set

List<Integer> evenList = someIntList.stream()

.filter(i -> i % 2 == 0)

.collect(**Collectors.toUnmodifiableList()**);

evenList.add(4);

**Parallel Full GC for G1**

The G1 garbage collector is the default one since JDK 9. However, the full GC for G1 used a single threaded mark-sweep-compact algorithm.

This has been **changed to the parallel mark-sweep-compact algorithm**in Java 10 effectively reducing the stop-the-world time during full GC.

**Experimental Java-Based JIT Compiler**

[Graal](https://github.com/oracle/graal/blob/master/compiler/README.md) is a dynamic compiler written in Java that integrates with the HotSpot JVM; it's focused on high performance and extensibility. It's also the basis of the experimental Ahead-of-Time (AOT) compiler introduced in JDK 9.

JDK 10 enables the Graal compiler, to be used as an experimental JIT compiler on the Linux/x64 platform.

**Java 10 LocalVariable Type-Inference**

Until Java 9, we had to mention the type of the local variable explicitly and ensure it was compatible with the initializer used to initialize it

**String** message = "Good bye, Java 9"; 🡺 Java 10 🡺 **var** message = "Hello, Java 10";

Map<Integer, String> map = **new** **HashMap**<>(); 🡺 java 10 🡺 **var** idToNameMap = **new** **HashMap**<Integer, String>();

**New Features in Java 11**

**Java 11 adds a few**[**new methods**](https://www.baeldung.com/java-11-string-api)**to the String class**: isBlank, lines, strip, stripLeading, stripTrailing, and repeat.

Let's see how we can make use of the new methods to extract non-blank, stripped lines from a multi-line string:

**String** multilineString = "Baeldung helps \n \n developers \n explore Java.";

List<String> lines = multilineString.lines()

.filter(line -> !line.isBlank())

.map(String::strip)

.collect(Collectors.toList());

assertThat(lines).containsExactly("Baeldung helps", "developers", "explore Java.");

**New File Methods**

Additionally, it's now easier to read and write Strings from files.

**We can use the new readString and writeString static methods from the *Files*class:**

**Path** filePath = Files.writeString(Files.createTempFile(tempDir, "demo", ".txt"), "Sample text");

**String** fileContent = Files.readString(filePath);

**Collection to an Array**

The java.util.Collection interface contains a new default toArray method which takes an IntFunction argument.

This makes it easier to create an array of the right type from a collection:

**List** sampleList = Arrays.asList("Java", "Kotlin");

String[] sampleArray = sampleList.toArray(String[]::**new**);

**A No-Op Garbage Collector**

A new garbage collector called Epsilon is available for use in Java 11 as an experimental feature.

It's called a No-Op (no operations) because it allocates memory but does not actually collect any garbage. Thus, Epsilon is applicable for simulating out of memory errors.

Obviously Epsilon won't be suitable for a typical production Java application; however, there are a few specific use-cases where it could be useful:

* Performance testing
* Memory pressure testing
* VM interface testing and
* Extremely short-lived jobs

In order to enable it, use the -XX:+UnlockExperimentalVMOptions -XX:+UseEpsilonGC flag.