**Java 8,9,10,11,17 Features – 2022**

**Java 8 Changes**

**Lambda Expression and Stream API**

**Default Methods - Interface Default and Static Methods**

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

**Optional – Use of Optional**

**Java IO improvements**

**Type Annotations - Later**

**Repeating Annotations – Later**

**Interface Default and Static 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** (In a typical design based on abstractions, where an interface has one or multiple implementations, if one or more methods are added to the interface, all the implementations will be forced to implement them too. Otherwise, the design will just break down.
* Default interface methods are an efficient way to deal with this issue. They **allow us to add new methods to an interface that are automatically available in the implementations**. Therefore, we don’t need to modify the implementing classes.)

**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. Good Example is given below.**

**public** **interface** **Vehicle** {

String **getBrand**();

String **speedUp**();

String **slowDown**();

**default** String **turnAlarmOn**() {

**return** "Turning the vehicle alarm on.";

}

**default** String **turnAlarmOff**() {

**return** "Turning the vehicle alarm off.";

}

}

public class Car implements Vehicle {  
  
 private String brand;  
  
 public Car(String brand) {  
 this.brand = brand;  
 }  
  
 *// constructors/getters* @Override  
 public String getBrand() {  
 return brand;  
 }  
  
 @Override  
 public String speedUp() {  
 return "The car is speeding up.";  
 }  
  
 @Override  
 public String slowDown() {  
 return "The car is slowing down.";  
 }  
}

public class Test1 {  
 public static void main(String[] args) {  
 Vehicle car = new Car("BMW");  
 System.*out*.println(car.getBrand());  
 System.*out*.println(car.speedUp());  
 System.*out*.println(car.slowDown());  
 System.*out*.println(car.turnAlarmOn());  
 System.*out*.println(car.turnAlarmOff());  
 }  
}

**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.**

**Another example**

List<Employee> empList = List.*of*(new Employee("John", 23),  
 new Employee("Vidya", 27), new Employee("Romano",17));  
**List<Employee> filterList = empList.stream().filter(e -> Test1.*isGreaterThan20*(e))  
 .collect(Collectors.*toList*());**  
**filterList.forEach(e -> System.*out*.println(e));**

The above can be written as

List<Employee> empList = List.*of*(new Employee("John", 23),  
 new Employee("Vidya", 27), new Employee("Romano",17));  
**List<Employee> filterList = empList.stream().filter(Test1::*isGreaterThan20*)  
 .collect(Collectors.*toList*());**  
**filterList.forEach(System.*out*::println);**

**Optional<T>**

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

**Java IO improvements**

Path filePath = Paths.*get*("pom.xml");  
try {  
 String contents = **Files.*readString*(filePath);** 🡸 Read the whole content as String  
 System.*out*.println("Contents: " + contents);  
 byte[] bytes = **Files.*readAllBytes*(filePath);** 🡸 Read all byte[]  
 System.*out*.println(new String(bytes));  
 List<String> lines = **Files.*readAllLines*(filePath);** 🡸 Reading all lines  
 lines.forEach(line -> System.*out*.println(line));  
 Stream<String> allLines = **Files.*lines*(filePath);** 🡸 Read line by line  
 allLines.forEach(System.*out*::println);  
} catch (IOException e) {  
 throw new RuntimeException(e);  
}

String text = "Text to save to file";

Path filePath = Paths.get("./fileName.txt")

Files.write(filePath, text.getBytes()); 🡸 Writing file

**New Features in Java 9**

**Try-with-resources**

**A New HTTP Client**

**Process API**

**Private Interface Methods**

**Diamond Syntax with Inner Anonymous Classes**

**Modular System – Jigsaw Project**

**Using *try-with-resources***

**try** (**PrintWriter** writer = **new** **PrintWriter**(**new** **File**("test.txt"))) {

writer.println("Hello World");

}

**Replacing *try*–*catch-finally* With *try-with-resources***

**try** (**Scanner** scanner = **new** **Scanner**(**new** **File**("test.txt"))) {

**while** (scanner.hasNext()) {

System.out.println(scanner.nextLine());

}

} **catch** (FileNotFoundException fnfe) {

fnfe.printStackTrace();

}

***try-with-resources* With Multiple Resources**

**try** (**Scanner** scanner = **new** **Scanner**(**new** **File**("testRead.txt"));

**PrintWriter** writer = **new** **PrintWriter**(**new** **File**("testWrite.txt"))) {

**while** (scanner.hasNext()) {

writer.print(scanner.nextLine());

}

}

***A Custom Resource With AutoCloseable***

To construct a custom resource that will be correctly handled by a *try-with-resources* block, the class should implement the *Closeable* or *AutoCloseable* interfaces and override the *close* method:

**public** **class** **MyResource** **implements** **AutoCloseable** {

@Override

**public** **void** **close**() **throws** Exception {

System.out.println("Closed MyResource");

}

}

**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!");

}

}

**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**

***Local Variable Type Inference - Java 10 LocalVariable Type-Inference***

***Parallel Full GC for G1***

**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>();

**Old Way**

public class LocalTypeVar {

public void explicitTypes() {

Person Roland = new Person("Roland", "Deschain");

Person Susan = new Person("Susan", "Delgado");

List<Person> persons =

List.of(Roland, Susan, Eddie, Detta, Jake);

for (Person person : persons) {

System.out.println(person.name + " - " + person.lastname);

}

}

}

**Implicit Typing with var**

public class LocalTypeVar {

public void varTypes() {

var Roland = new Person("Roland", "Deschain");

var Susan = new Person("Susan", "Delgado");

var persons = List.of(Roland, Susan, Eddie, Detta, Jake);

for (var person : persons) {

System.out.println(person.name + " - " + person.lastname);

}

}

}

**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.

**New Features in Java 14**

**Switch Expressions**

**Old Way of Switch Statements**

public static void main(String[] args) {  
 int days = 0;  
 Month month = Month.*APRIL*;  
  
 switch (month) {  
 case *JANUARY*, *MARCH*, *MAY*, *JULY*, *AUGUST*, *OCTOBER*, *DECEMBER* :  
 days = 31;  
 break;  
 case *FEBRUARY* :  
 days = 28;  
 break;  
 case *APRIL*, *JUNE*, *SEPTEMBER*, *NOVEMBER* :  
 days = 30;  
 break;  
 default:  
 throw new IllegalStateException();  
 }  
}

**Using Switch Expressions**

public static void main(String[] args) {

int days = 0;

Month month = Month.APRIL;

days = switch (month) {

case JANUARY, MARCH, MAY, JULY, AUGUST, OCTOBER, DECEMBER -> 31;

case FEBRUARY -> 28;

case APRIL, JUNE, SEPTEMBER, NOVEMBER -> 30;

default -> throw new IllegalStateException();

};

}

**The yield Keyword**

public static void main(String[] args) {

int days = 0;

Month month = Month.APRIL;

days = switch (month) {

case JANUARY, MARCH, MAY, JULY, AUGUST, OCTOBER, DECEMBER -> {

System.out.println(month);

yield 31;

}

case FEBRUARY -> {

System.out.println(month);

yield 28;

}

case APRIL, JUNE, SEPTEMBER, NOVEMBER -> {

System.out.println(month);

yield 30;

}

default -> throw new IllegalStateException();

};

}

**New Features in Java 15**

Text Blocks

**Example Without Using Text Blocks**

public static void main(String[] args) {

System.out.println(

"<!DOCTYPE html>\n" +

"<html>\n" +

" <head>\n" +

" <title>Example</title>\n" +

" </head>\n" +

" <body>\n" +

" <p>This is an example of a simple HTML " +

"page with one paragraph.</p>\n" +

" </body>\n" +

"</html>\n");

}

**Example of Using Text Blocks**

public static void main(String[] args) {

System.out.println(

"""

<!DOCTYPE html>

<html>

<head>

<title>Example</title>

</head>

<body>

<p>This is an example of a simple HTML

page with one paragraph.</p>

</body>

</html>

"""

);

}

**New Features in Java 16**

Pattern Matching of instanceof

Java Record Classes

**Example Without Pattern Matching**

public class PatternMatching {

public static double priceOld(Vehicle v) {

if (v instanceof Car) {

Car c = (Car) v;

return 10000 - c.kilomenters \* 0.01 -

(Calendar.getInstance().get(Calendar.YEAR) -

c.year) \* 100;

} else if (v instanceof Bicycle) {

Bicycle b = (Bicycle) v;

return 1000 + b.wheelSize \* 10;

} else throw new IllegalArgumentException();

}

}

**Using Pattern Matching**

public class PatternMatching {

public static double price(Vehicle v) {

if (v instanceof Car c) {

return 10000 - c.kilomenters \* 0.01 -

(Calendar.getInstance().get(Calendar.YEAR) -

c.year) \* 100;

} else if (v instanceof Bicycle b) {

return 1000 + b.wheelSize \* 10;

} else throw new IllegalArgumentException();

}

}

**Records**

Refer to Record document

**New Features in Java 17**

Sealed Classes

The sealed class allows us to make class effectively final for everyone except explicitly mentioned classes.

**Constraints**

Several constraints have to be met for the sealed class to work:

* Permitted subclasses must be accessible by the sealed class at compile time
* Permitted subclasses must directly extend the sealed class
* Permitted subclasses must have one of the following modifiers:
  + final
  + sealed
  + non-sealed
* Permitted subclasses must be in the same Java module