Java 9 to 17

Let’s truly upgrade!

Akshita Chawla | 29th Aug

Although we have upgraded to Java 17 in our pom files but our code looks like this

So what are we going to learn today

- Brief overview of Java versioning
- Java 17 - LTS : Sealed Classes
- Java 16 : Pattern Matching instance of
- Java 16 : Records
- Java 15 : Text Blocks
- Java 14 : Switch Expressions
- Java 10 : Local Variable Type Inference
- Java 11 -LTS : Local Variable Type in Lambda Expressions
- Java 9 : Private Interface Methods
- Java 9 : Diamond Operator for Anonymous Inner Class
- Java 9 : Try With Resources Improvement

Another new version? What's this LTS now?

In June 2018, just , Oracle announced a change to the release cadence model for Java SE.

Rather than having a major release planned for every two to three years (which would often become three to four years), a new six-month feature-release-train model would be used: Every three years, a release would be designated as *Long-Term Support* (LTS) and receive quarterly security, stability, and performance updates only.

This pattern borrowed shamelessly from the Mozilla Firefox release model but tweaked it to be more aligned with the requirements of a development platform.

LTS version

- Java 8
- Java 11
- Java 17
- Java 21 (Releasing on 19th Sept)

Six Month Release Cycle

- March
- September

Java 17 : Sealed Classes

Let's say you have written a "beautiful" algorithm that you want only share with your team, but your code is also exposed to external users.
What do you ensure only "VIP" members get excess to your algorithm?

The `final` modifier on a class doesn't allow anyone to extend it. What about when we want to extend a class but only allow it for some classes?

This is where Java 17 comes into play with sealed classes. The sealed class allows us to make class effectively final for everyone except explicitly mentioned classes.

```
public sealed class Vehicle permits Bicycle, Car {...}
```

We added a `sealed` modifier to our `Vehicle` class, and we had to add the `permits` keyword with a list of classes that we allow to extend it.
After this change, we are still getting errors from the compiler.

There is one more thing that we need to do here.

We need to add `final`, `sealed`, or `non-sealed` modifiers to classes that will extend our class.

```
public final class Bicycle extends Vehicle {...}
```

Java 16

With features like
Pattern matching of instanceof
&
Records,
Java 16 has made programmers' lives easier by reduction of boilerplate code

Java 16 : Pattern Matching of instanceof

The pattern matching for *instanceof* operator avoids the boilerplate code to type test and cast to a variable more concisely

Before Java 16

```
1   Customer customer = null;
2   String customerName;
3
4   if(customer instanceof PersonalCustomer)
5   {
6     PersonalCustomer pc = (PersonalCustomer) customer; //Redundant casting
7     customerName = String.join(" ", pc.getFirstName(),
8       pc.getMiddleName(), pc.getLastName());
9   }
10  else if(customer instanceof BusinessCustomer)
11  {
12    BusinessCustomer bc = (BusinessCustomer) customer; //Redundant casting
13    customerName = bc.getLegalName();
14  }
15
```

We don't need line 6 and 12 anymore

After Java 16

```
1   Customer customer = null;
2   String customerName;
3
4   if(customer instanceof PersonalCustomer pc)
5   {
6     customerName = String.join(" ", pc.getFirstName(),
7       pc.getMiddleName(), pc.getLastName());
8   }
9   else if(customer instanceof BusinessCustomer bc)
10  {
11    customerName = bc.getLegalName();
12  }
13
```

Pattern is a combination of

- A predicate that can be applied to a target
- A set of binding variables that are extracted from the target only if the predicate successfully applies to it

The assignment of pattern variable happens only when the predicate test is true.

Java 16 : Pattern Matching of instanceof

Scope of Variables

```
if (!(obj instanceof String s)) {
    //System.out.println(s);  //compiler error - cannot access 's' here
}
```

The print statement will execute only when the predicate fails, so this statement is not even compiled.

The variable is available in the enclosing *if* block, and we cannot access it outside it.

```
Object obj = new Object();

if (obj instanceof String s) {
  System.out.println(s);    //can use 's' here
} else if (obj instanceof Number n) {
  //System.out.println(s);    //can not use 's' here
}
//System.out.println(s);    //can not use 's' here
```

```
//Allowed to use s in complex condition
if (obj instanceof String s && s.startsWith("a")) {
  System.out.println(s);
}

//compiler error
if (obj instanceof String s || s.startsWith("a")) {
  System.out.println(s);
}
```

If we are writing complex conditional statements in the *if*-statement then **we can use the pattern variable only with** `&&` ***(AND)*** **operator** because the variable will be accessed only when the predicate has been tested to be true. **We cannot use the variable with** `||` ***(OR)*** **operator** because the predicate may be tested *false* in a few cases, and then we will be exceptions of different kinds.

Java 16 : Record

If I asked how many PoJos have you written in your entire life, the answer always be “Hell lot of it”

Java has had a bad reputation for boilerplate code.
Lombok allowed us to stop worrying about getters, setters, etc.
Java 16 finally introduced records to remove a lot of boilerplate code

Records provide a compact syntax for declaring classes which are plain immutable data carriers

Before Java 16

```java
public class Person {
    private final String name;
    private final String gender;
    private final int age;

    public Person(String name, String gender, int age) {
        this.name = name;
        this.gender = gender;
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public String getGender() {
        return gender;
    }

    public int getAge() {
        return age;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o)
            return true;
        if (o == null || getClass() != o.getClass())
            return false;
        Person person = (Person) o;
        return age == person.age &&
                Objects.equals(name, person.name) &&
                Objects.equals(gender, person.gender);
    }

    @Override
    public int hashCode() {
        return Objects.hash(name, gender, age);
    }

    @Override
    public String toString() {
        return "Person{" +
                "name='" + name + '\'' +
                ", gender='" + gender + '\'' +
                ", age=" + age +
                '}';
    }
}
```

New Way

```java
public record Person(String name, String gender, int age) {}
```

```
$ javap Person.class
Compiled from "Person.java"
public final class com.logicbig.example.Person extends java.lang.Record {
  public com.logicbig.example.Person(java.lang.String, java.lang.String, int);
  public final java.lang.String toString();
  public final int hashCode();
  public final boolean equals(java.lang.Object);
  public java.lang.String name();
  public java.lang.String gender();
  public int age();
}
```

Java 15 : Text Blocks

Text block is an improvement on formatting String variables.
From Java 15, we can write a String that spans through several lines as regular text.

Before Java 15

```
public class TextBlocks {

    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");
    }
}
```

After Java 15

```
public class TextBlocks {

    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>
                """
        );
    }
}
```

Java 15 : Text Blocks

A text block begins with three double-quote characters followed by a line terminator.

You can't put a text block on a single line, nor can the contents of the text block follow the three opening double-quotes without an intervening line terminator.

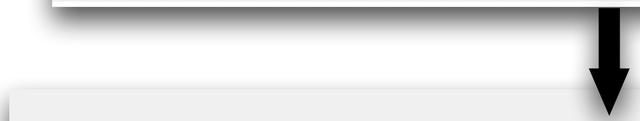
```
// ERROR
String name = """Pat Q. Smith""";

// ERROR
String name = """red
                green
                blue
                """;

// OK
String name = """
    red
    green
    blue
    """;
```

How can we use it in our code

```
String query = "SELECT `EMP_ID`, `LAST_NAME` FROM `EMPLOYEE_TB`\n" +
               "WHERE `CITY` = 'INDIANAPOLIS'\n" +
               "ORDER BY `EMP_ID`, `LAST_NAME`;\n";
```



```
String query = """
               SELECT `EMP_ID`, `LAST_NAME` FROM `EMPLOYEE_TB`
               WHERE `CITY` = 'INDIANAPOLIS'
               ORDER BY `EMP_ID`, `LAST_NAME`;
               """;
```

Java 14 : Switch Expressions

Switch expressions allowed us to omit break calls inside every case block.
It helps with the readability of the code and better understanding

switch label “case L ->”, the switch block code looks clearer, more concise and more readable

Before Java 14

```
public class SwitchExpression {

    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();
        }
    }
}
```

After Java 14

```
public class SwitchExpression {

    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();
        };
    }
}
```

Java 14 : Switch Expressions

If the code of a case is a block of code, you can use the ***yield*** keyword to return the value for the switch block.

```
public class SwitchExpression {

    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();
        };
    }
}
```

Java 10 : Local Variable Type Inference

We declare local variables with non-null initializers with the var identifier, which can help you write code that's easier to read.

Java always needed explicit types on local variables.
The var type allows us to omit type from the left-hand side of our statements.

```
URL url = new URL("http://www.oracle.com/");
URLConnection conn = url.openConnection();
Reader reader = new BufferedReader(
    new InputStreamReader(conn.getInputStream()));
```

```
var url = new URL("http://www.oracle.com/");
var conn = url.openConnection();
var reader = new BufferedReader(
    new InputStreamReader(conn.getInputStream()));
```

var is a reserved type name, not a keyword,
which means that existing code that uses var as a variable, method, or package name is not affected.
However, code that uses var as a class or interface name is affected and the class or interface needs to be renamed.

Java 11 : Local Variable Type in Lambda Expressions

```
(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.

Java 11 introduced an improvement to the previously mentioned local type inference.
This allows us to use `var` inside lambda expressions.

```
(@Nonnull var s1, @Nullable var s2) -> s1 + s2
```

Limitations

```
(var s1, s2) -> s1 + s2  // Compilation error, we cannot use var for some parameters and skip for others

(var s1, String s2) -> s1 + s2 // Compilation error, we cannot mix var with explicit types
```

Java 9 : Private Interface Methods

In **Java 8,** we can provide method implementation in Interfaces using **Default and Static methods.**
However we cannot create private methods in Interfaces.

To **avoid redundant code and more re-usability,** Oracle Corp is going to introduce private methods in Java SE 9 Interfaces.
From Java SE 9 onwards, we can write private and private static methods too in an interface using a ‘private’ keyword.

These private methods are like other class private methods only, there is no difference between them.

```
public interface TempI {

    public abstract void mul(int a, int b);

public default void
    add(int a, int b)
    {
// private method inside default method
        sub(a, b);

 // static method inside other non-static method
        div(a, b);
        System.out.print("Answer by Default method = ");
        System.out.println(a + b);
    }

    public static void mod(int a, int b)
    {
        div(a, b); // static method inside other static method
        System.out.print("Answer by Static method = ");
        System.out.println(a % b);
    }

    private void sub(int a, int b)
    {
        System.out.print("Answer by Private method = ");
        System.out.println(a - b);
    }

    private static void div(int a, int b)
    {
        System.out.print("Answer by Private static method = ");
        System.out.println(a / b);
    }
}
```

Java 9 : Try With Resources Improvement

Try with resources blocks are introduced from Java 7. In these blocks, resources used in try blocks are auto-closed. No need to close the resources explicitly.

But, Java 7 try with resources has one drawback.

It requires resources to be declared locally within try block. It doesn't recognize resources declared outside the try block.

That issue has been resolved in Java 9.

We can now use final or even effectively final variables inside a try-with-resources block:

```
try (Scanner scanner = new Scanner(new File("testRead.txt"));
    PrintWriter writer = new PrintWriter(new File("testWrite.txt"))) {
    // omitted
}
```

Java 7

```
final Scanner scanner = new Scanner(new File("testRead.txt"));
PrintWriter writer = new PrintWriter(new File("testWrite.txt"))
try (scanner;writer) {
    // omitted
}
```

Java 9

Java 9 : Diamond Operator for Anonymous Inner Class

Diamond operator was introduced as a new feature in java SE 7.
The purpose of diamond operator is to avoid redundant code by leaving the generic type in the right side of the expression.
Java 7 allowed us to use diamond operator in normal classes but it didn't allow us to use them in anonymous inner classes.
Java 9 improved the use of diamond operator and allows us to use the diamond operator with anonymous inner classes.

Java 7

```
abstract class MyClass<T>{
    abstract T add(T num, T num2);
}
public class JavaExample {
    public static void main(String[] args) {
        MyClass<Integer> obj = new MyClass<>() {
            Integer add(Integer x, Integer y) {
                return x+y;
            }
        };
        Integer sum = obj.add(100,101);
        System.out.println(sum);
    }
}
```

```
$javac JavaExample.java
JavaExample.java:7: error: cannot infer type arguments for MyClass
        MyClass obj = new MyClass<>() {
                                  ^
  reason: cannot use '<>' with anonymous inner classes
  where T is a type-variable:
    T extends Object declared in class MyClass
1 error
```

Java 9

```
abstract class MyClass<T>{
    abstract T add(T num, T num2);
}
public class JavaExample {
    public static void main(String[] args) {
        MyClass<Integer> obj = new MyClass<>() {
            Integer add(Integer x, Integer y) {
                return x+y;
            }
        };
        Integer sum = obj.add(100,101);
        System.out.println(sum);
    }
}
```