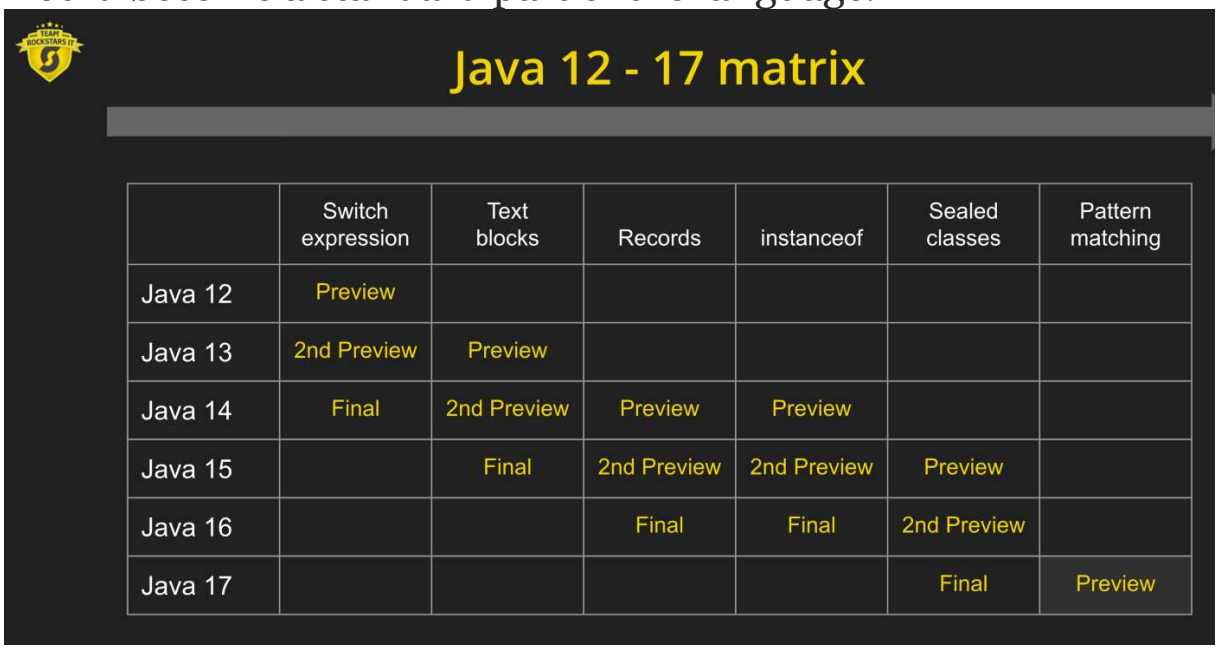
In the middle of September, Java 17 got released. This is the new LTS (Long Term Support) version, replacing Java 11. In this article I am going over the changes in the language that happened since Java 11. Java is definitely still on the move and in more than one direction.

Below is an image that details the changes I am going to discuss here and in what version they were added. Most changes were first introduced as a preview version for which you had to opt-in. The language designers gathered feedback from the community and then released a 2nd preview in the next release (which would be 6 months later). Finally, in yet another next release it would become a standard part of the language.



	Switch expression	Text blocks	Records	instanceof	Sealed classes	Pattern matching
Java 12	Preview					
Java 13	2nd Preview	Preview				
Java 14	Final	2nd Preview	Preview	Preview		
Java 15		Final	2nd Preview	2nd Preview	Preview	
Java 16			Final	Final	2nd Preview	
Java 17					Final	Preview

Switch expressions

Since the beginning of times the switch statement has been a part of the Java language. It had its limitations and quirks however and now switch functions add a modern style to the

language. Basically, there were two new styles added, dubbed switch labelled rule and switch labelled statement group.

```
    private void run(int value) {
        String answer = switch (abs(value)) {
            case 0 -> "Zero";
            case 1,2,3,4,5  -> "Five or less";
            case 42 -> "Answer to life,the universe and everything";
            default -> "Not sure, but more than six";
        };
        System.out.println(answer);
    }
```

The above code shows the new syntax where you can have multiple case labels and the resulting value after the -> . This value will be assigned to the answer variable. This is a **switch labelled rule**.

The second variant allows for a block of statements after a matching label, hence the name **switch labelled statement group**. It uses the new keyword **yield** to return a value:

```
private void runAnother(int value) {
        String answer = switch (abs(value)) {
            case 0:
                System.out.println("Value is zero");
                yield "Zero";
            case 1,2,3,4,5:
                System.out.println("Value is between 1 and 5");
                yield "Five or less";
            case 42:
                System.out.println("42!!!");
                yield "Answer to life, the universe and everything";
            default:
                System.out.println("Another value");
                yield "Not sure, but mor than six";
```

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

Text blocks

A very welcome addition to the language are text blocks. Before If you had a String spanning multiple lines you had to concatenate them via a + and had to start the next line with double quotes (“) again. Also, using double quotes within the string meant that you had to escape it. This has finally been fixed with the arrival of text blocks.

```
private void run() throws Exception {
   var writer = new BufferedWriter(new FileWriter("demo.txt"));
   writer.write("""
        <html>
           <body>
              <p>
                 <div>Finally some text with "
                 </div>
              </p>
           </body>
        </html>""")
   writer.close();
}
```

Text blocks start with a **triple double quote** and a newline character. Note the placement of the closing triple double quotes at the end. When placed on the same line as the text (as in the example above) no new line character is include, otherwise it is.

Records

Records are easily the most anticipated and discussed new feature in the Java language since the addition of streams and lambdas in Java 8. And for good reason: They offer a huge improvement in readability and conciseness and they add this funky “modern language” feeling to Java.

```
public record SimpleRecord(String name, boolean isStriped) { }
```

This is the most simplest form of a record. You use the keyword **record** and after the name you specify a list of instance variables. Note that the variables are effectively final, meaning that after the constructor has completed their values cannot be changed anymore. Hence there are no setter/mutator methods nor can you add them yourself.

The Java compiler will automatically generate the constructor, getters for the instance variables, hashcode, equals and toString methods. The getters do not follow the JavaBean convention (e.g. getName in the above example) but have the same name as the instance variable. So you would access the value of the variable ‘name’ via *name()*

Java Records also define the concept of a **compact constructor**. Basically it tells you “I know what constructor arguments I can expect (as they are defined in the record definition header), so you don’t need to tell me”. Compact constructor can be used to validate the incoming arguments for example.

```
public SimpleRecord {
    requireNonNull(name, "Name cannot be null");
}
```

Notice in the example above that we do not specify the incoming parameter values and also observe that we do not need to assign these parameter values to the instance variables. Under the hood the compiler generates code to take care of those tasks.

You cannot add any additional instance variables other than those defined in the record definition header. But you can add static variables, static methods and static initialisers. You can also add your own instance methods.

While minimising code is one of the benefits it is not the only reason to use records. Using them as lightweight data transfer objects is another. But arguably the most important reason is enhanced security and control during data deserialisation. With regular classes a (de)serialisation framework like Jackson could use a number of techniques to bypass using the setters and thus circumventing any validation that might be in place there. With records you are ensured that your validation logic in the constructor will be invoked as there is no other way to create a record. Creating records via reflection for example is not allowed and will throw an exception.

instanceof pattern matching

With some enhancements to the instanceof operator Java started dipping its toes into the lake of pattern matching.

Pattern matching, like so many phrases in IT, is an overloaded terminology. It this context it is not related to regular expressions. It means a technique where we have a predicate that is applied to an object and if predicate matches some values of the tested object they are extracted into pattern variables.

Let's look at an example:

```
if (animal instanceof Cat c) {
   c.speak();
   c.jump();
}
```

Here, *animal* is the object we want to examine. The test we apply is 'is this object an instance of type Cat'. If this condition returns true, the object is assigned to the pattern variable 'c' which itself is of type Cat. Any casting is implicit, so this reads as the traditional approach:

```
Cat c = (Cat)animal;
```

While this first approach to pattern matching may look quite trivial it marks an important direction the language has taken. Later in this article we will see some additional pattern matching improvements with regard to switch expressions.
Some additional JEPs are already defined that extend pattern matching into the realms of records and arrays.

Sealed classes

Sealed classes are a new feature that let classes and interfaces set restrictions on the classes and interfaces that are allowed to

subclass or extend them. This can massively help correctness and completeness when you define your object model. And there is the added benefit that it allows the compiler (and the developer) to reason about exhaustiveness.

```
public sealed interface Feline permits Tiger, Puma, Ocelot {
}
```

In the above code fragment we define an interface Feline. The **sealed** keyword indicates that we want to limit subclassing. The classes/interfaces that are allowed to subclass the Feline interface are named after the **permits** keyword. In this case Tiger, Puma and Ocelot.

These classes themselves can introduce sealing as well. They can either be sealed as well, be final or non-sealed. Let's have a look at each of the possible options.

```
public sealed class Tiger implements Feline permits HouseTiger {}
```

The Tiger subclass is sealed itself as well. Tiger itself allows one other class to subclass it, HouseTiger.

A second possibility is that you want no other classes to subclass your class. This can be achieved by marking the class as **final** and omitting the permits keyword, as shown below:

```
public final class Ocelot implements Feline {}
```

A third and final option is stop limiting subclassing, basically returning to the inheritance possibilities that we already have without sealed classes. In code, it looks like this, notice the use of the **non-sealed** keyword:

```
public non-sealed class HouseTiger extends Tiger {}
```

Pattern match for switch

The final new language feature that was added to Java 17 is pattern matching for switch expressions. Note that this is a preview version. This is the second occurrence of pattern matching in the language after it has been introduced via the instanceof operator.

In its most simple form, pattern matching is performed on the **type** of variable that is supplied. Compare this to a classical switch expression/statement where the *value* is used. Also note that there is a special handling for the **null** values. Normally, a null value would lead to a NullPointerException (NPE) so it would require an additional check. Thanks to the special handling of the null value this is no longer required. Finally, also notice that the required casting is again performed automatically by the compiler.

```
    private String printValue(Object obj) {
        return switch (obj) {
            case Integer i -> String.format("It is an integer with value
%d", i);
            case Long l -> String.format("It is a Long with value %d", l);
```

```
            case String s -> String.format("It is a String with value %s",
s);
            case null -> new String("You can't pass in a null value!");
            default -> String.format("Dunno the type, but the value is %s",
obj.toString());
        };
    }
```

There are two additional patterns defined for a switch. The first one is called a **guarded pattern** where an additional condition can be added to a case expression.

```
    private String printValue(Object obj) {
        return switch (obj) {
            case null -> new String("You can't pass in a null value!");
            case String s && s.length() > 10 -> String.format("Long String
with value %s", s);
            case String s -> String.format("Not so long String with value
%s", s);
            default -> String.format("Dunno the type, but the value is %s",
obj.toString());
        };
    }
```

In the example above, a distinction can be made between strings with a length of up to 10 characters and strings with more than 10 characters.

Finally, there is the **parenthesised pattern**. Basically, this is when you use additional parenthesis to prevent an invalid outcome of a pattern predicate. See this example:

Conclusion

Since the last LTS, Java has seen many language improvements. While some of these are more standalone there is definitely a coherence discoverable between switch pattern matching, sealed classes and records.

Oracle has recently decided that it will release an LTS version every two years from now on. This used to be three years. So the next LTS version will be Java 21.

With so many new and exciting features added to the language you should feel encourages to upgrade your existing projects to the latest Java version.

The code examples in this article plus some additional examples can be found in my GitHub repository.