Using Pattern Matching

Introducing Pattern Matching

Pattern matching is a feature that is still being worked on. Some elements of this feature have been released as final feat

If you want to learn more about pattern matching and provide feedback, then you need to visit the Amber project page. Th

If you are new to pattern matching, the first thing you may have in mind is pattern matching in regular expressions. If this

Regular expressions are a form of pattern matching that has been created to analyze strings of characters. It is a good ar

Let us write the following code.

```
String sonnet = "From fairest creatures we desire increase,\n" +
        "That thereby beauty's rose might never die,\n" +
        "But as the riper should by time decease\n" +
        "His tender heir might bear his memory:\n" +
        "But thou, contracted to thine own bright eyes,\n" +
        "Feed'st thy light's flame with self-substantial fuel,\n" +
        "Making a famine where abundance lies,\n" +
        "Thyself thy foe, to thy sweet self too cruel.\n" +
        "Thou that art now the world's fresh ornament,\n" +
        "And only herald to the gaudy spring,\n" +
        "Within thine own bud buriest thy content,\n" +
        "And, tender churl, mak'st waste in niggardly.\n" +
        "Pity the world, or else this glutton be,\n" +
        "To eat the world's due, by the grave and thee.";

Pattern pattern = Pattern.compile("\\bflame\\b");
Matcher matcher = pattern.matcher(sonnet);
while (matcher.find()) {
    String group = matcher.group();
    int start = matcher.start();
    int end = matcher.end();
    System.out.println(group + " " + start + " " + end);
}
```

This code takes the first sonnet of Shakespeare as a text. This text is analyzed with the regular expression `\bflame\b`. Th

You can do much more things with regular expression. It is outside the scope of this tutorial. If you want to learn more ab

If you run this code, it will print the following:

```
flame 233 238
```

This result tells you that there is a single occurrence of `flame` between the index 233 and the index 238 in the sonnet.

Pattern matching with regular expression works in this way:

1. it matches a given *pattern*; `flame` is this example and matches it to a text
2. then it gives you information on the place where the pattern has been matched.

There are three notions that you need to keep in mind for the rest of this tutorial:

1. What you need to match; this is called the *matched target*. Here it is the sonnet.
2. What you match against; this is called the *pattern*. Here the regular expression `flame`.
3. The result of the matching; here the start index and the end index.

These three elements are the fundamental elements of pattern matching.

Pattern Matching for Instanceof

Matching Any Object to a Type with Instanceof

There are several ways of extending pattern matching. The first one that we cover is called *Pattern matching for instanceo*

Let us extend the example of the previous section to the `instanceof` use case. For that, let us consider the folowing exan

```
public void print(Object o) {
    if (o instanceof String s){
        System.out.println("This is a String of length " + s.length());
    } else {
        System.out.println("This is not a String");
    }
}
```

Let us describe the three elements we presented there.

The *matched target* is any object of any type. It is the left-hand side operand of the `instanceof` operator: `o`.

The *pattern* is a type followed by a variable declaration. It is the right hand-side of the `instanceof`. The type can be a clas

The result of the matching is a new reference to the *matched target*. This reference is put in the variable that is declared a

In our example, the variable `o` is the element you need to match; it is your *matched target*. The *pattern* is the `String s` dec

This special syntax where you can define a variable along with the type declared with the `instanceof` is a new syntax add

The pattern `String s` is called a *type pattern*, because it checks the type of the matched target. Note that because the typ

```
public void print(Object o) {
    if (o instanceof CharSequence cs) {
        System.out.println("This is a CharSequence of length " + s.length());
    }
}
```

Using the Pattern Variable

The compiler allows you to use the variable `s` wherever it makes sense to use it. The `if` branch is the first scope that com

The following code checks if `object` is an instance of the `String` class, and if it is a non-empty string. You can see that it

```
public void print(Object o) {
    if (o instanceof String s && !s.isEmpty()) {
        int length = s.length();
        System.out.println("This object is a non-empty string of length " + length);
    } else {
        System.out.println("This object is not a string.");
    }
```

```
}
```

There are cases where your code checks for the real type of a variable, and if this type is not the one you expect, then you

```
public void print(Object o) {
    if (!(o instanceof String)) {
        return;
    }
    String s = (String)o;
    // do something with s
}
```

Starting with Java SE 16, you can write this code in that way, leveraging pattern matching for `instanceof`:

```
public void print(Object o) {
    if (!(o instanceof String s)) {
        return;
    }

    System.out.println("This is a String of length " + s.length());
}
```

The `s` pattern variable is available oustide of the `if` statement, as long as your code leaves the method from the `if` branc

There are cases where the compiler can tell if the matching fails. Let us consider the following example:

```
Double pi = Math.PI;
if (pi instanceof String s) {
    // this will never be true!
}
```

The compiler knows that the `String` class is final. So there is no way that the variable `pi` can be of type `String`. The com

Writing Cleaner Code with Pattern Matching for Instanceof

There are many places where using this feature will make your code much more readable.

Let us create the following `Point` class, with an `equals()` method. The `hashCode()` method is omitted here.

```
public class Point {
    private int x;
    private int y;

    public boolean equals(Object o) {
        if (!(o instanceof Point)) {
            return false;
        }
        Point point = (Point) o;
        return x == point.x && y == point.y;
    }

    // constructor, hashCode method and accessors have been omitted
}
```

This is the classic way of writing an `equals()` method; it could have been generated by an IDE.

You can rewrite this `equals()` method with the following code that is leveraging the pattern matching for `instanceof` fea

```
public boolean equals(Object o) {
    return o instanceof Point point &&
```

```
            x == point.x &&
            y == point.y;
}
```

Pattern Matching for Switch

Extending Switch Expressions to Use Type Patterns for Case Labels

Pattern Matching for Switch is a final feature of the JDK 21. It was presented as a preview feature in Java SE 17, 18, 19 ar

Pattern Matching for Switch uses switch statements or expressions. It allows you to match a *matched target* to several *pa*

In this case the *matched target* is the selector expression of the switch. There are several *patterns* in such a feature; each

Let us consider the following code.

```
Object o = ...; // any object
String formatted = null;
if (o instanceof Integer i) {
    formatted = String.format("int %d", i);
} else if (o instanceof Long l) {
    formatted = String.format("long %d", l);
} else if (o instanceof Double d) {
    formatted = String.format("double %f", d);
} else {
    formatted = String.format("Object %s", o.toString());
}
```

You can see that it contains three *type patterns*, one for each if statement. Pattern matching for switch allows to write this

```
Object o = ...; // any object
String formatter = switch(o) {
    case Integer i -> String.format("int %d", i);
    case Long l    -> String.format("long %d", l);
    case Double d  -> String.format("double %f", d);
    default        -> String.format("Object %s", o.toString());
};
```

Not only does pattern matching for switch makes your code more readable; it also makes it more performant. Evaluating

So far it is not an extension of pattern matching itself; it is a new feature of the switch, that accepts a type pattern as a ca

In its current version, the switch expression accepts the following for the case labels:

1. the following numeric types: `byte`, `short`, `char`, and `int` (`long` is not accepted)
2. the corresponding wrapper types: `Byte`, `Short`, `Character` and `Integer`
3. the type `String`
4. enumerated types.

Pattern matching for switch adds the possibility to use type patterns for the case labels.

Using Guarded Patterns

In the case of Pattern Matching for `instanceof`, you already know that the pattern variable created if the matched target r

```java
Object object = ...; // any object
if (object instanceof String s && !s.isEmpty()) {
    int length = s.length();
    System.out.println("This object is a non-empty string of length " + length);
}
```

This works well in an if statement, because the argument of the statement is a boolean type. In switch expressions, case

```java
Object o = ...; // any object
String formatter = switch(o) {
    // !!! THIS DOES NOT COMPILE !!!
    case String s && !s.isEmpty() -> String.format("Non-empty string %s", s);
    case Object o                 -> String.format("Object %s", o.toString());
};
```

It turns out that the *pattern matching for switch* has been extended to allow for a boolean expression to be added after the

```java
Object o = ...; // any object
String formatter = switch(o) {
    case String s when !s.isEmpty() -> String.format("Non-empty string %s", s);
    default                         -> String.format("Object %s", o.toString());
};
```

This extended case label is called a *guarded case label*. The expression `String s when !s.isEmpty()` is such a guarded

Record Pattern

A *record* is a special type of immutable class, written as such, introduced in Java SE 16. You can visit our Record page to

A record pattern is a special kind of pattern, published as a final feature in Java SE 21. It was available as a preview featu

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

This information enables a notion called *record deconstruction*, use in record pattern matching. The following code is a fir

```java
Object o = ...; // any object
if (o instanceof Point(int x, int y)) {
    // do something with x and y
}
```

The *target operand* is still the `o` reference. It is matched to a *record pattern*: `Point(int x, int y)`. This pattern declares t

A record pattern is built with the name of the record, `Point` in this example, and one type pattern per component of that re

A record pattern is built on the same model as the canonical constructor of a record. Even if you create other constructor

```java
record Point(int x, int y) {
    Point(int x) {
        this(x, 0);
    }
}

Object o = ...; // any object
// !!! THIS DOES NOT COMPILE !!!
if (o intanceof Point(int x)) {

```

```
11
    }
```

Record pattern supports type inference. The type of the components you use to write your pattern can be inferred with `va`

Because the matching of each component is actually a type pattern, you can match a type that is an extension of the actu

Here is a first example where you can ask the compiler to infer the real type of your binding variable.

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

Object o == ...; // any object
if (o instanceof Point(var x, var y)) {
    // x and y are of type double
}
```

On the following example, you can switch on the type of the component of the `Box` record.

```
record Box(Object o) {}

Object o = ...; // any object
switch (o) {
    case Box(String s)  -> System.out.println("Box contains the string: " + s);
    case Box(Integer i) -> System.out.println("Box contains the integer: " + i);
    default -> System.out.println("Box contains something else");
}
```

As it is the case for `instanceof`, you cannot check for a type that is not possible. Here, the type `Integer` cannot extend th

```
record Box(CharSequence o) {}

Object o = ...; // any object
switch (o) {
    case Box(String s)  -> System.out.println("Box contains the string: " + s);
    // !!! THE FOLLOWING LINE DOES NOT COMPILE !!!
    case Box(Integer i) -> System.out.println("Box contains the integer: " + i);
    default -> System.out.println("Box contains something else");
}
```

Record patterns do not support boxing nor unboxing. So the following code is not valid.

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

Object o = ...; // any object
// !!! DOES NOT COMPILE !!!
if (o instanceof Point(int x, int y)) {
}
```

One last point: record pattern support nesting, so you can write the following code.

```
record Point(double x, double y) {}
record Circle(Point center, double radius) {}

Object o = ...; // any object
if (o instanceof Circle(Point(var x, var y), var radius)) {
    // Do something with x, y and radius
}
```

More Patterns

Pattern matching is now supported by three elements of the Java language, as final feature or as a preview feature:

- the `instanceof` keyword,
- the `switch` statement and expression,
- and the extends `for` loop.

They all support two kinds of patterns: *type patterns* and *record patterns*.

There is more to come in the near future. More elements of the Java language could be modified and more kind of patter

Last update: *December 21, 2022*