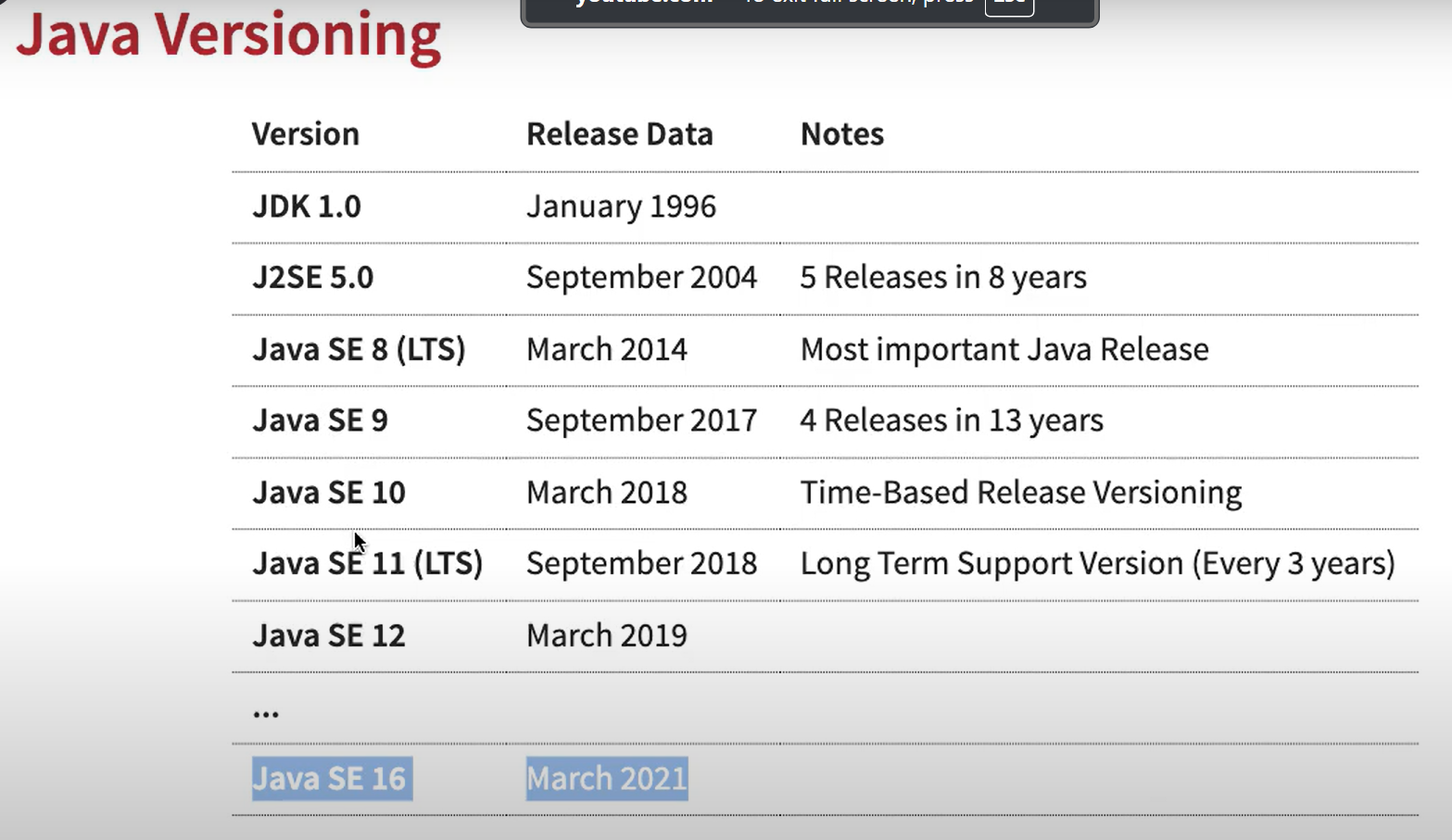
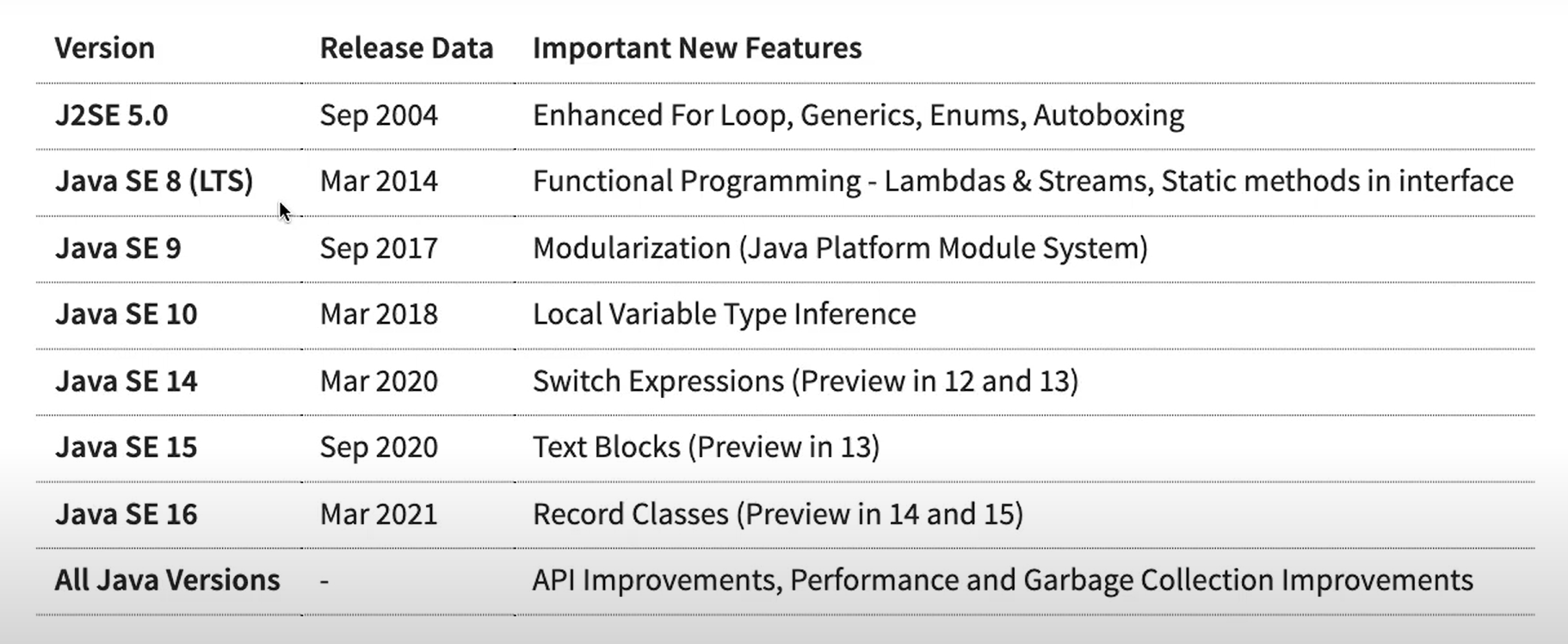
Java 8(mar-2014),11(sep-2018),17(sep-2021), 21(sep-2023) – Long Time Support

8 - Where functional programming introduced

Latest version 23(mar-2025)



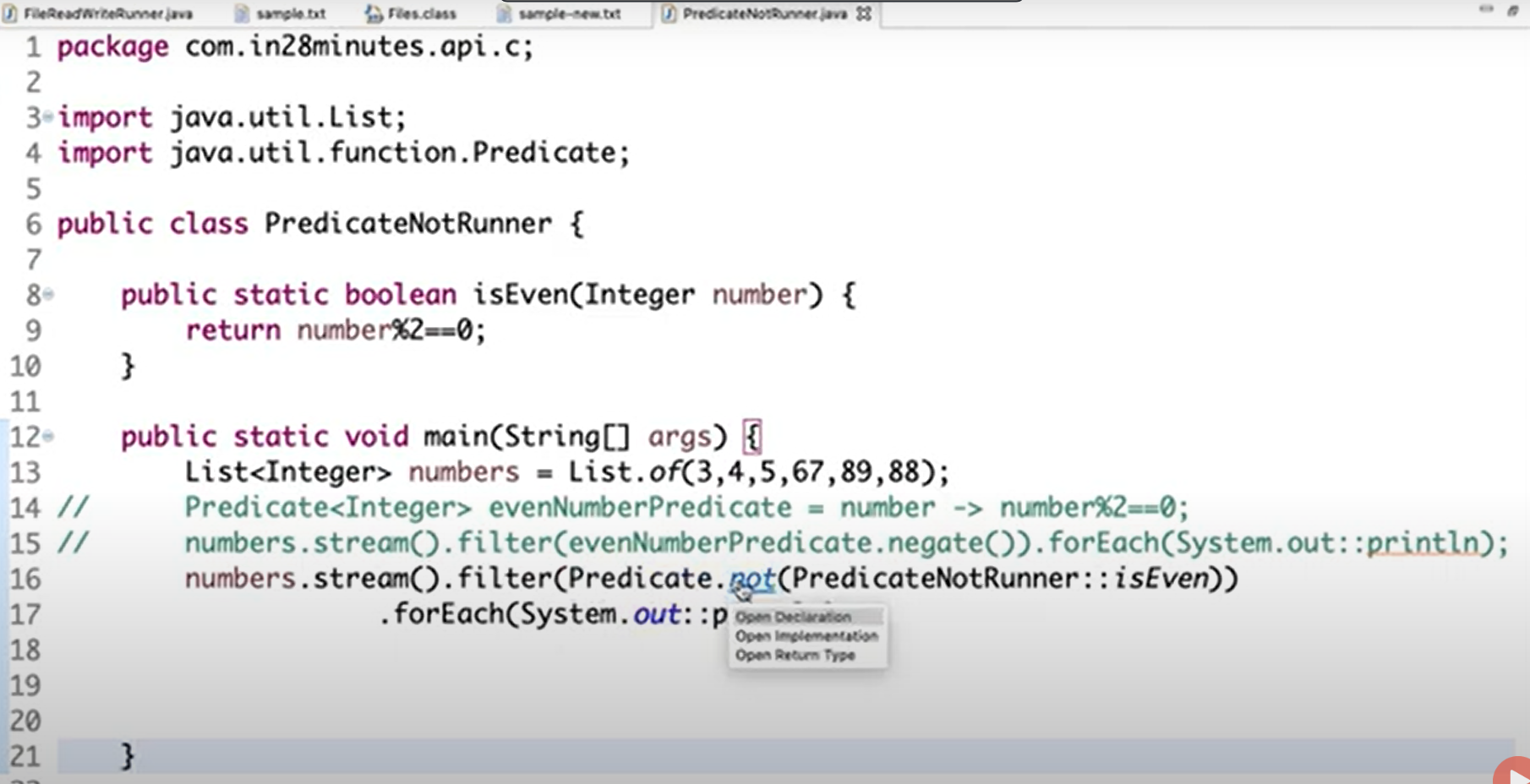


Of()- java 9

copyOf() – java10

readString/writeString – java11

Predicate.not()



String utility methods –

Java 9 🡪

* Private Method in Interface
* Try-with-resource improvements

[Java Language Changes](https://docs.oracle.com/en/java/javase/21/language/java-language-changes.html#GUID-6459681C-6881-45D8-B0DB-395D1BD6DB9B)

# Java 16 🡪

[New Features in Java 16 | Baeldung](https://www.baeldung.com/java-16-new-features#records-jep-395)

Records 🡪 a restricted form of a class

* Introduced in Java 14. But have some incremental changes in java 16.
* a record is a special type of class declaration aimed at reducing the boilerplate code.
* Introduced with the aim to be used as a fast way to create data carrier classes (POJO. DTOs).
* ***Records are a better choice than classes in situations where you are primarily storing data and not defining any behaviour.***

In simple class objects we need to create constructor, getter, setter(). If we want to use this object with data structures like HashMap we need to implement equals(),hashcode(), toString() to print as String. But with Record it can be reduced.

Public **record** Employee(int id, String firstName, String lastName){}

All the instance fields are written as parameters. The constructor, getter methods, toString(), equals(), and hashCode() are generated by the Java compiler during compile time. One thing to note here is that records do not provide setter methods, as it is expected that the value to instance variables is provided while creating the object.

Employee e = new Employee(1001, “parul”, “jain”);

System.out.printn(e.id()+” ”+ e.firstName()+” ”+e.lastName());

System.out.println(e.toString());

**Some more Properties of Records**

* Records are implicitly final.
* You can use nested classes and interfaces inside a record.
* You can have nested records too, which will implicitly be static.
* A record can implement interfaces.
* You can create a generic record class.
* It is possible to use local record classes (since Java SE 15).
* Records are serializable.

That is not all that a record can do. Records also provide us the capability to:

* **Create our own constructors.** In records, you can create a parameterized constructor, which calls the default constructor with the provided parameters inside its body. You can also create compact constructors which are like default constructors with the twist that you can add some extra functionality such as checks inside the constructor body.
* **Create instance methods.** Like any other class, you can create and call instance methods for the record class.
* **Create static fields.** Records restrict us to write the instance variables only as parameters but enable the use of static variables and static methods.

Public record Employee(int id, String fname, String lname){

Static int empToken;

Public Employee(int id, String fname){

This(id, fname, null);

}

Public void getFullName(){

System.out.println(fname() + lname());

}

Public static int generateToken(){

Return ++empToken;

}

// Constructor 1 of this class

// Compact Constructor

public Employee

{

if (id < 100) {

throw new IllegalArgumentException(

"Employee Id cannot be below 100.");

}

if (firstName.length() < 2) {

throw new IllegalArgumentException(

"First name must be 2 characters or more.");

}

}

}

# Java 21 🡪

[Java 21 Features (LTS): Practical Examples and Insights](https://howtodoinjava.com/java/java-21-new-features/)

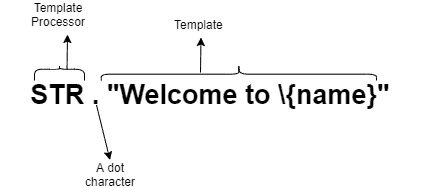
## String Template 🡪

it will allow you to mix the expression and code together,

String name = "parul Jain";  
System.out.println("Hello...."+ name + "!!!, welcome");

// But now with this update you might be able to write this in an easier way, something like

System.out.println(STR."Hello \{name} !!!, Welcome");



At runtime, when the template processor evaluates the template expression, it combines the literal text in the template with the values of the embedded expression to produce the result.

## Pattern Matching

### [Record Patterns](https://docs.oracle.com/en/java/javase/21/language/record-patterns.html) 🡪

You can use a record pattern to test whether a value is an instance of a record class type (see [Record Classes](https://docs.oracle.com/en/java/javase/21/language/records.html#GUID-6699E26F-4A9B-4393-A08B-1E47D4B2D263)) and, if it is, to recursively perform pattern matching on its component values.

For background information about record patterns, see [JEP 440](https://openjdk.java.net/jeps/440).

A record pattern is a construct that allows us to match values against a record type and bind variables to the corresponding components of the record.

1. **Basic Usage**: You can use a record pattern to test if an object is an instance of a record class and extract its components. For example:

Point p = new Point(1,5);

if (p instanceof Point (int x, int y)) {//Record Pattern

//if(p instanceOf Point p)// type pattern  
 System.*out*.println("x point is "+ x +" y point is "+ y);  
}

**Note:**The null value does not match any record pattern.

1. **Nested Record Patterns**: Record patterns can be nested to decompose complex records. For example:

Point p = new Point(1,5);  
Line line = new Line(new Point(1, 4), new Point(3,5));

if (line instanceof Line(Point(int x1, int y1), Point(int x2, int y2))){  
 System.*out*.println(x1 + x2);  
 System.*out*.println(y1 + y2);  
}

enum Color { RED, GREEN, BLUE }

record ColoredPoint(Point p, Color c) {}

record ColoredRectangle(ColoredPoint upperLeft, ColoredPoint lowerRight) {}

static void printXCoordOfUpperLeftPointWithPatterns(ColoredRectangle r) {

if (r instanceof ColoredRectangle(

ColoredPoint(Point(var x, var y), var upperLeftColor),

var lowerRightCorner)) {

System.out.println("Upper-left corner: " + x);

}

}

You can do the same for parameterized records. The compiler infers the types of the record pattern's type arguments and pattern variables. In the following example, the compiler infers Box(Box(var s)) as Box<Box<String>>(Box(String s)).

static void nestedBox(Box<Box<String>> bo) {

// Box(Box(var s)) is inferred to be Box<Box<String>>(Box(var s))

if (bo instanceof Box(Box(var s))) {

System.out.println("String " + s);

}

}

1. **Type Inference/Using var**: You can use var in record patterns to let the compiler infer the types of the components:

//the null value doesn’t match any record pattern. We can replace the type of the variables with var.  
// In that specific case, the compiler will infer the type for us:  
if(p instanceof Point(var x, var y)){  
 System.*out*.println("x point is "+ x +" y point is "+ y);  
}

You can use var in the record pattern's component list. In the following example, the compiler infers that the pattern variables x and y are of type double:

static void printAngleFromXAxis(Object obj) {

if (obj instanceof Point(var x, var y)) {

System.out.println(Math.toDegrees(Math.atan2(y, x)));

}

}

The compiler can infer the type of the type arguments for record patterns in all constructs that accept patterns: switch statements, switch expressions, and instanceof expressions.

The following example is equivalent to printBoxContents. The compiler infers its type argument and pattern variable: Box(var s) is inferred as Box<String>(String s)

static void printBoxContentsAgain(Box<String> bo) {

if (bo instanceof Box(var s)) {

System.out.println("Box contains: " + s);

}

}

1. **Generic Records**: Record patterns also support generic records. If a record class is generic, then you can explicitly specify the type arguments in a record pattern. For example:

record Wrapper<T>(T value, String description) {}  
if (obj instanceof Wrapper<String>(String value, String description)) {  
 System.*out*.println("Value: " + value + ", Description: " + description);  
  
}}

**Example 2**

record Box<T>(T t) { }

static void printBoxContents(Box<String> bo) {

if (bo instanceof Box<String>(String s)) {

System.out.println("Box contains: " + s);

}

}

1. **Switch cases:**

switch (line){  
 case Line(Point p1, Point(var x, var y)):  
 System.*out*.println(x + y);  
 default:  
 System.*out*.println(0+0);  
}

### Pattern Matching for the instanceof Operator

Pattern matching involves testing whether an object has a particular structure, then extracting data from that object if there's a match. You can already do this with Java; however, pattern matching introduces new language enhancements that enable you to conditionally extract data from objects with code that's more concise and robust.

For background information about pattern matching for the instanceof operator, see [JEP 394](https://openjdk.java.net/jeps/394).

Consider the following code that calculates the perimeter of certain shapes:

public interface Shape {

public static double getPerimeter(Shape s) throws IllegalArgumentException {

if (s instanceof Rectangle) {

Rectangle r = (Rectangle) s;

return 2 \* r.length() + 2 \* r.width();

} else if (s instanceof Circle) {

Circle c = (Circle) s;

return 2 \* c.radius() \* Math.PI;

} else {

throw new IllegalArgumentException("Unrecognized shape");

}

}

}

public class Rectangle implements Shape {

final double length;

final double width;

public Rectangle(double length, double width) {

this.length = length;

this.width = width;

}

double length() { return length; }

double width() { return width; }

}

public class Circle implements Shape {

final double radius;

public Circle(double radius) {

this.radius = radius;

}

double radius() { return radius; }

}

The method getPerimeter performs the following:

1. A test to determine the type of the Shape object
2. A conversion, casting the Shape object to Rectangle or Circle, depending on the result of the instanceof operator
3. A destructuring, extracting either the length and width or the radius from the Shape object

Pattern matching enables you to remove the conversion step by changing the second operand of the instanceof operator with a type pattern, making your code shorter and easier to read:

public static double getPerimeter(Shape shape) throws IllegalArgumentException {

if (s instanceof Rectangle r) {

return 2 \* r.length() + 2 \* r.width();

} else if (s instanceof Circle c) {

return 2 \* c.radius() \* Math.PI;

} else {

throw new IllegalArgumentException("Unrecognized shape");

}

}

### Type Matching in switch 🡪

**The main goal of this feature is to allow patterns in switch case labels and improve the expressiveness of switch statements and expressions.** Besides, there is also an enhancement to handle NullPointerException by allowing a null case label.

class Account{  
 double getBalance(){  
 return 0;  
 }  
}  
  
class SavingsAccount extends Account {  
 double getSavings() {  
 return 100;  
 }  
}  
class TermAccount extends Account {  
 double getTermAccount() {  
 return 1000;  
 }  
}  
class CurrentAccount extends Account {  
 double getCurrentAccount() {  
 return 10000;  
 }  
}

Before Java 21, we can use the below code to get the balance:

static double getBalanceWithOutSwitchPattern(Account account) {

double balance = 0;

if(account instanceof SavingsAccount sa) {

balance = sa.getSavings();

}

else if(account instanceof TermAccount ta) {

balance = ta.getTermAccount();

}

else if(account instanceof CurrentAccount ca) {

balance = ca.getCurrentAccount();

}

return balance;

}

The above code isn’t very expressive as we have a lot of noise with the *if*–*else* cases. With Java 21, we can leverage patterns in *case* labels to write the same logic more concisely:

static double getBalanceWithSwitchCase(Account account){  
 switch (account) {  
 case null:  
 throw new RuntimeException("Ooops, account balance is null");  
 case SavingsAccount sa:  
 return sa.getSavings();  
 case TermAccount ta:  
 return ta.getTermAccount();  
 case CurrentAccount ca:  
 return ca.getCurrentAccount();  
 default:  
 return account.getBalance();  
 }  
}

### [Pattern Matching for switch Expressions and Statements](https://docs.oracle.com/en/java/javase/21/language/pattern-matching-switch.html#GUID-5C1E3BBD-B12D-493E-AC19-D27ABC628D76)) ([JEP 441](https://openjdk.org/jeps/441))

Initially introduced in JDK 17.

A switch statement transfers control to one of several statements or expressions, depending on the value of its selector expression. In earlier releases, the selector expression must evaluate to a number, string or enum constant, and case labels must be constants. However, in this release, the **selector expression can be any reference type or an int type but not a long, float, double, or boolean type**, and **case labels can have patterns**. Consequently, a switch statement or expression can test whether its selector expression matches a pattern, which offers more flexibility and expressiveness compared to testing whether its selector expression is exactly equal to a constant.

#### Selector Expression Type

**The type of a selector expression can either be an integral primitive type or any reference type, such as in the previous examples**. The following switch expression matches the selector expression obj with type patterns that involve a class type, an enum type, a record type, and an array type:

record Point(int x, int y) { }

enum Color { RED, GREEN, BLUE; }

...

static void typeTester(Object obj) {

switch (obj) {

case null -> System.out.println("null");

case String s -> System.out.println("String");

case Color c -> System.out.println("Color with " + c.values().length + " values");

case Point p -> System.out.println("Record class: " + p.toString());

case int[] ia -> System.out.println("Array of int values of length" + ia.length);

default -> System.out.println("Something else");}}

#### When Clauses

You can add a Boolean expression right after a pattern label with a when clause. This is called a *guarded pattern label*. The Boolean expression in the when clause is called a *guard*. A value matches a guarded pattern label if it matches the pattern and, if so, the guard also evaluates to true. Consider the following example:

static void test(Object obj) {

switch (obj) {

case String s:

if (s.length() == 1) {

System.out.println("Short: " + s);

} else {

System.out.println(s);

}

break;

default:

System.out.println("Not a string");

}

}

You can move the Boolean expression s.length == 1 right after the the case label with a when clause:

static void test(Object obj) {

switch (obj) {

case String s when s.length() == 1 -> System.out.println("Short: " + s);

case String s -> System.out.println(s);

default -> System.out.println("Not a string");

}

}

The first pattern label (which is a guarded pattern label) matches if obj is both a String and of length 1. The second patten label matches if obj is a String of a different length.

A guarded patten label has the form *p* when *e* where *p* is a pattern and *e* is a Boolean expression. The scope of any pattern variable declared in *p* includes *e*.

Example 2, suppose we need to process an input string that contains a simple “*Yes*” or “*No*“:

static String processInputOld(String input) {  
 String output = null;  
 switch(input) {  
 case null -> output = "Oops, input is null";  
 case String s -> {  
 if("Yes".equalsIgnoreCase(s)) {  
 output = "Input is Yes";  
 }  
 else if("No".equalsIgnoreCase(s)) {  
 output = "Input is No";  
 }  
 else {  
 output = "Try Again";  
 }  
 }  
 }  
 return output;  
}

Again, we can see that writing *if*–*else*logic can get ugly. Instead, in Java 21, we can use *when* clauses along with *case* labels to match the label’s value against an expression:

static String processInputNew(String input) {  
 String output = null;  
 switch(input) {  
 case null -> output = "Oops, input is null";  
 case String s **when** "Yes".equalsIgnoreCase(s)->  
 output = "Input is Yes";  
 case String s **when** "No".equalsIgnoreCase(s)->  
 output = "Input is No";  
 default ->{  
 output = "Try Again";}  
 }  
 return output;  
}

#### Qualified enum Constants as case Constants

You can use qualified enum constants as case constants in switch expressions and statements.

Consider the following switch expression whose selector expression is an enum type:

public enum Standard { SPADE, HEART, DIAMOND, CLUB }

static void determineSuitStandardDeck(Standard d) {

switch (d) {

case SPADE -> System.out.println("Spades");

case HEART -> System.out.println("Hearts");

case DIAMOND -> System.out.println("Diamonds");

default -> System.out.println("Clubs");

}

}

In the following example, the type of the selector expression is an interface that's been implemented by two enum types. Because the type of the selector expression isn't an enum type, this switch expression uses guarded patterns instead:

sealed interface CardClassification permits Standard, Tarot {}

public enum Standard implements CardClassification

{ SPADE, HEART, DIAMOND, CLUB }

public enum Tarot implements CardClassification

{ SPADE, HEART, DIAMOND, CLUB, TRUMP, EXCUSE }

static void determineSuit(CardClassification c) {

switch (c) {

case Standard s when s == Standard.SPADE -> System.out.println("Spades");

case Standard s when s == Standard.HEART -> System.out.println("Hearts");

case Standard s when s == Standard.DIAMOND -> System.out.println("Diamonds");

case Standard s -> System.out.println("Clubs");

case Tarot t when t == Tarot.SPADE -> System.out.println("Spades or Piques");

case Tarot t when t == Tarot.HEART -> System.out.println("Hearts or C\u0153ur");

case Tarot t when t == Tarot.DIAMOND -> System.out.println("Diamonds or Carreaux");

case Tarot t when t == Tarot.CLUB -> System.out.println("Clubs or Trefles");

case Tarot t when t == Tarot.TRUMP -> System.out.println("Trumps or Atouts");

case Tarot t -> System.out.println("The Fool or L'Excuse");

}

}

However, switch expressions and statements allow qualified enum constants, so you could rewrite this example as follows:

static void determineSuitQualifiedNames(CardClassification c) {

switch (c) {

case Standard.SPADE -> System.out.println("Spades");

case Standard.HEART -> System.out.println("Hearts");

case Standard.DIAMOND -> System.out.println("Diamonds");

case Standard.CLUB -> System.out.println("Clubs");

case Tarot.SPADE -> System.out.println("Spades or Piques");

case Tarot.HEART -> System.out.println("Hearts or C\u0153ur");

case Tarot.DIAMOND -> System.out.println("Diamonds or Carreaux");

case Tarot.CLUB -> System.out.println("Clubs or Trefles");

case Tarot.TRUMP -> System.out.println("Trumps or Atouts");

case Tarot.EXCUSE -> System.out.println("The Fool or L'Excuse");

}

}

Therefore, you can use an enum constant when the type of the selector expression is not an enum type provided that the enum constant's name is qualified and its value is assignment-compatible with the type of the selector expression.

Points to be remember🡪

* *Pattern Label Dominance*

It's possible that many pattern labels could match the value of the selector expression. To help predictability, **the labels are tested in the order that they appear in the switch block**. In addition, **the compiler raises an error if a pattern label can never match because a preceding one will always match first**. The following example results in a compile-time error:

static void error(Object obj) {

switch(obj) {

case CharSequence cs ->

System.out.println("A sequence of length " + cs.length());

case String s -> // error: this case label is dominated by a preceding case label

System.out.println("A string: " + s);

default -> { break; }

}

}

The first pattern label case CharSequence cs *dominates* the second pattern label case String s because every value that matches the pattern String s also matches the pattern CharSequence cs but not the other way around. It's because String is a subtype of CharSequence.

A pattern label can dominate a constant label. These examples cause compile-time errors:

static void error2(Integer value) {

switch(value) {

case Integer i ->

System.out.println("Integer: " + i);

case -1, 1 -> // Compile-time errors for both cases -1 and 1:

// this case label is dominated by a preceding case label

System.out.println("The number 42");

default -> { break; }

}

}

enum Color { RED, GREEN, BLUE; }

static void error3(Color value) {

switch(value) {

case Color c ->

System.out.println("Color: " + c);

case RED -> // error: this case label is dominated by a preceding case label

System.out.println("The color red");

}

}

**Note:**

**Guarded pattern labels don't dominate constant labels.** For example:

static void testInteger(Integer value) {

switch(value) {

case Integer i when i > 0 ->

System.out.println("Positive integer");

case 1 ->

System.out.println("Value is 1");

case -1 ->

System.out.println("Value is -1");

case Integer i ->

System.out.println("An integer");

}

}

Although the value 1 matches both the guarded pattern label case Integer i when i > 0 and the constant label case 1, the guarded pattern label doesn't dominate the constant label. Guarded patterns aren't checked for dominance because they're generally undecidable. ***Consequently, you should order your case labels so that constant labels appear first, followed by guarded pattern labels, and then followed by nonguarded pattern labels:***

static void testIntegerBetter(Integer value) {

switch(value) {

case 1 ->

System.out.println("Value is 1");

case -1 ->

System.out.println("Value is -1");

case Integer i when i > 0 ->

System.out.println("Positive integer");

case Integer i ->

System.out.println("An integer");

}

}

* ***Type Coverage in switch Expressions and Statements***

As described in [Switch Expressions](https://docs.oracle.com/en/java/javase/21/language/switch-expressions-and-statements.html#GUID-BA4F63E3-4823-43C6-A5F3-BAA4A2EF3ADC), **the switch blocks of switch expressions and switch statements, which use pattern or null labels, must be exhaustive. This means that for all possible values, there must be a matching switch label**. The following switch expression is not exhaustive and generates a compile-time error. Its type coverage consists of the subtypes of String or Integer, which doesn't include the type of the selector expression, Object:

static int coverage(Object obj) {

return switch (obj) { // Error - not exhaustive

case String s -> s.length();

case Integer i -> i;

};

}

However, the type coverage of the case label **default** is all types, so the following example compiles:

static int coverage(Object obj) {

return switch (obj) {

case String s -> s.length();

case Integer i -> i;

**default -> 0;**

};

}

**The compiler takes into account whether the type of a selector expression is a sealed class.** The following switch expression compiles. It doesn't need a default case label because its type coverage is the classes A, B, and C, which are the only permitted subclasses of S, the type of the selector expression:

sealed interface S permits A, B, C { }

final class A implements S { }

final class B implements S { }

record C(int i) implements S { } // Implicitly final

...

static int testSealedCoverage(S s) {

return switch (s) {

case A a -> 1;

case B b -> 2;

case C c -> 3;

};

}

The compiler can also determine the type coverage of a switch expression or statement if the type of its selector expression is a generic sealed class. The following example compiles. The only permitted subclasses of interface I are classes A and B. However, because the selector expression is of type I<Integer>, the switch block requires only class B in its type coverage to be exhaustive:

sealed interface I<T> permits A, B {}

final class A<X> implements I<String> {}

final class B<Y> implements I<Y> {}

static int testGenericSealedExhaustive(I<Integer> i) {

return switch (i) {

// Exhaustive as no A case possible!

case B<Integer> bi -> 42;

};

}

The type of a switch expression or statement's selector expression can also be a generic record. As always, a switch expression or statement must be exhaustive. The following example doesn't compile. No match for a Pair exists that contains two values, both of type A:

record Pair<T>(T x, T y) {}

class A {}

class B extends A {}

static void notExhaustive(Pair<A> p) {

switch (p) {

// error: the switch statement does not cover all possible input values

case Pair<A>(A a, B b) -> System.out.println("Pair<A>(A a, B b)");

case Pair<A>(B b, A a) -> System.out.println("Pair<A>(B b, A a)");

}

}

The following example compiles. Interface I is sealed. Types C and D cover all possible instances:

record Pair<T>(T x, T y) {}

sealed interface I permits C, D {}

record C(String s) implements I {}

record D(String s) implements I {}

static void exhaustiveSwitch(Pair<I> p) {

switch (p) {

case Pair<I>(I i, C c) -> System.out.println("C = " + c.s());

case Pair<I>(I i, D d) -> System.out.println("D = " + d.s());

}

}

If a switch expression or statement is exhaustive at compile time but *not* at run time, then a MatchException is thrown. This can happen when a class that contains an exhaustive switch expression or statement has been compiled, but a sealed hierarchy that is used in the analysis of the switch expression or statement has been subsequently changed and recompiled. Such changes are *migration incompatible* and may lead to a MatchException being thrown when running the switch statement or expression. Consequently, you need to recompile the class containing the switch expression or statement.

Consider the following two classes ME and Seal:

Copy

class ME {

public static void main(String[] args) {

System.out.println(switch (Seal.getAValue()) {

case A a -> 1;

case B b -> 2;

});

}

}

Copy

sealed interface Seal permits A, B {

static Seal getAValue() {

return new A();

}

}

final class A implements Seal {}

final class B implements Seal {}

The switch expression in the class ME is exhaustive and this example compiles. When you run ME, it prints the value 1. However, suppose you edit Seal as follows and compile this class and *not* ME:

sealed interface Seal permits A, B, C {

static Seal getAValue() {

return new A();

}

}

final class A implements Seal {}

final class B implements Seal {}

final class C implements Seal {}

When you run ME, it throws a MatchException:

Exception in thread "main" java.lang.MatchException

at ME.main(ME.java:3)

***Inference of Type Arguments in Record Patterns***

The compiler can infer the type arguments for a generic record pattern in all constructs that accept patterns: switch statements, instanceof expressions, and enhanced for statements.

In the following example, the compiler infers MyPair(var s, var i) as MyPair<String, Integer>(String s, Integer i):

record MyPair<T, U>(T x, U y) { }

static void recordInference(MyPair<String, Integer> p){

switch (p) {

case MyPair(var s, var i) ->

System.out.println(s + ", #" + i);

}

}

See [Record Patterns](https://docs.oracle.com/en/java/javase/21/language/record-patterns.html#GUID-7623D3AD-4141-4914-A384-60C65BD0C010) for more examples of inference of type arguments in record patterns.

#### Scope of Pattern Variable Declarations

As described in the section [Pattern Matching for the instanceof Operator](https://docs.oracle.com/en/java/javase/21/language/pattern-matching-instanceof.html#GUID-843060B5-240C-4F47-A7B0-95C42E5B08A7), the scope of a pattern variable is the places where the program can reach only if the instanceof operator is true:

public static double getPerimeter(Shape shape) throws IllegalArgumentException {

if (shape instanceof Rectangle s) {

// You can use the pattern variable s of type Rectangle here.

} else if (shape instanceof Circle s) {

// You can use the pattern variable s of type Circle here

// but not the pattern variable s of type Rectangle.

} else {

// You cannot use either pattern variable here.

}

}

In a switch expression or statement, the scope of a pattern variable declared in a case label includes the following:

* The when clause of the case label:

static void test(Object obj) {

switch (obj) {

case Character c when c.charValue() == 7:

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

break;

default:

break;

}

}

}

The scope of pattern variable c includes the when clause of the case label that contains the declaration of c.

* The expression, block, or throw statement that appears to the right of the arrow of the case label:

static void test(Object obj) {

switch (obj) {

case Character c -> {

if (c.charValue() == 7) {

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

}

System.out.println("Character, value " + c.charValue());

}

case Integer i ->

System.out.println("Integer: " + i);

default -> {

break;

}

}

}

The scope of pattern variable c includes the block to the right of case Character c ->. The scope of pattern variable i includes the println statement to the right of case Integer i ->.

* The switch-labeled statement group of a case label:

static void test(Object obj) {

switch (obj) {

case Character c:

if (c.charValue() == 7) {

System.out.print("Ding ");

}

if (c.charValue() == 9) {

System.out.print("Tab ");

}

System.out.println("character, value " + c.charValue());

default:

// You cannot use the pattern variable c here:

break;

}

}

The scope of pattern variable c includes the case Character c statement group: the two if statements and the println statement that follows them. The scope doesn't include the default statement group even though the switch statement can execute the case Character c statement group, fall through the default label, and then execute the default statement group.

**Note:**

You will get a compile-time error if it's possible to fall through a case label that declares a pattern variable. The following example doesn't compile:

static void test(Object obj) {

switch (obj) {

case Character c:

if (c.charValue() == 7) {

System.out.print("Ding ");

}

if (c.charValue() == 9) {

System.out.print("Tab ");

}

System.out.println("character");

case Integer i: // Compile-time error

System.out.println("An integer " + i);

default:

System.out.println("Neither character nor integer");

}

}

If this code were allowed, and the value of the selector expression, obj, were a Character, then the switch statement can execute the case Character c statement group and then fall through the case Integer i label, where the pattern variable i would have not been initialized.

#### Null case Labels

switch expressions and switch statements used to throw a NullPointerException if the value of the selector expression is null. Currently, to add more flexibility, a null case label is available:

static void test(Object obj) {

switch (obj) {

case null -> System.out.println("null!");

case String s -> System.out.println("String");

default -> System.out.println("Something else");

}

}

This example prints null! when obj is null instead of throwing a NullPointerException.

You may not combine a null case label with anything but a default case label. The following generates a compiler error:

static void testStringOrNull(Object obj) {

switch (obj) {

// error: invalid case label combination

case null, String s -> System.out.println("String: " + s);

default -> System.out.println("Something else");

}

}

However, the following compiles:

static void testStringOrNull(Object obj) {

switch (obj) {

case String s -> System.out.println("String: " + s);

case null, default -> System.out.println("null or not a string");

}

}

If a selector expression evaluates to null and the switch block does not have null case label, then a NullPointerException is thrown as normal. Consider the following switch statement:

String s = null;

switch (s) {

case Object obj -> System.out.println("This doesn't match null");

// No null label; NullPointerException is thrown

// if s is null

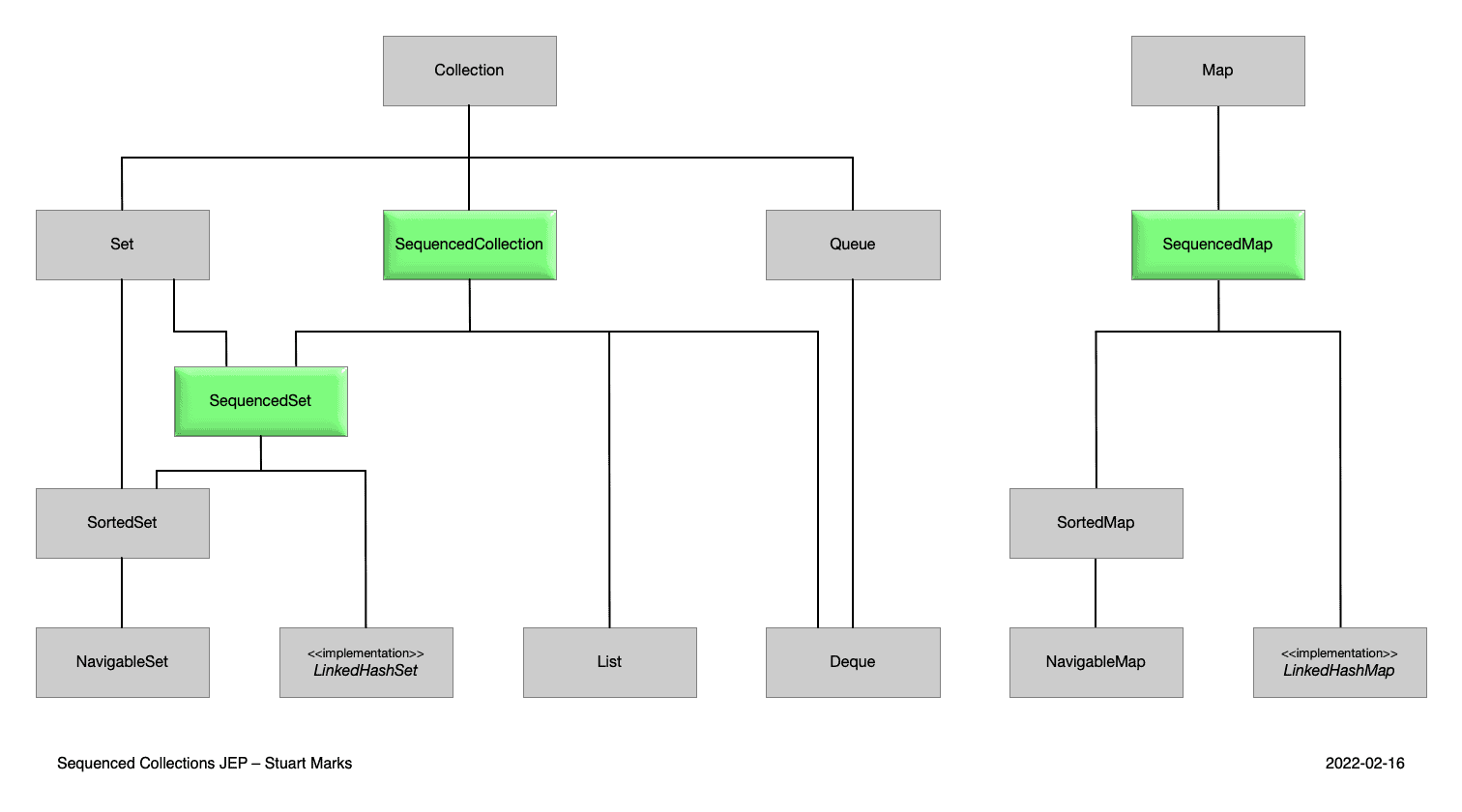
}

Although the pattern label case Object obj matches objects of type String, this example throws a NullPointerException. The selector expression evaluates to null, and the switch expression doesn't contain a null case label.

### [Unnamed Patterns and Variables](https://docs.oracle.com/en/java/javase/21/language/unnamed-variables-and-patterns.html)

## The New Java Collection Hierarchy/SequencedCollection interfaces

**This new feature introduces three new interfaces for sequenced collections, sequenced sets, and sequenced maps, which are added to the existing hierarchy of collections:**



### SequencedCollection

**A sequenced collection is a *Collection* whose elements have a defined encounter order.** The new *SequencedCollection* interface provides methods to add, retrieve, or remove elements at both ends of the collection, along with a method to get a reverse-ordered view of the collection.

**interface** **SequencedCollection**<E> **extends** **Collection**<E> {

// new method

SequencedCollection<E> **reversed**();

// methods promoted from Deque

**void** **addFirst**(E); **void** **addLast**(E); E **getFirst**();

E **getLast**(); E **removeFirst**(); E **removeLast**();

}

All methods, except *reversed()*, are default methods, provide a default implementation, and are promoted from *Deque*. The *reversed()* method provides a reversed-order view of the original collection. Also, any modifications to the original collection are visible in the reversed view.

The *add\*()* and *remove\*()* methods are optional and throw an *UnsupportedOperationException* in their default implementation, primarily to support the case of unmodifiable collections and collections with an already defined sorting order. The *get\*()* and *remove\*()* methods throw *NoSuchElementException* if the collection is empty.

### SequencedSet

**A sequenced set can be defined as a specialized *Set* which functions as a *SequencedCollection*, ensuring the absence of duplicate elements.** The *SequencedSet* interface extends*SequencedCollection* and overrides its *reversed()* method. The only difference is that the return type of*SequencedSet.reversed()* is *SequencedSet*.

**interface** **SequencedSet**<E> **extends** **Set**<E>, SequencedCollection<E> {

SequencedSet<E> **reversed**();

}

### SequencedMap

**A sequenced map is a *Map* whose entries have a defined encounter order.** The *SequencedMap* does not extend *SequencedCollection* and provides its own methods to manipulate elements at either end of the collection.

**interface** **SequencedMap**<K, V> **extends** **Map**<K, V> {

// new methods

SequencedMap<K, V> **reversed**();

SequencedSet<K> **sequencedKeySet**();

SequencedCollection<V> **sequencedValues**();

SequencedSet<Entry<K, V>> **sequencedEntrySet**();

V **putFirst**(K, V);

V **putLast**(K, V);

// methods promoted from NavigableMap

Entry<K, V> **firstEntry**();

Entry<K, V> **lastEntry**();

Entry<K, V> **pollFirstEntry**();

Entry<K, V> **pollLastEntry**();

}

Similar to *SequencedCollection*, *put\*()* methods throw *UnsupportedOperationException* for unmodifiable maps or maps with an already defined sorting order. Also, calling one of the methods promoted from *NavigableMap* on an empty map leads to throwing a *NoSuchElementException*.

## Virtual Threads

Virtual threads were initially introduced to the Java language as a preview feature in Java 19 and further refined in Java 20. Java 21 introduced some new changes.

**Virtual threads are lightweight threads with the purpose of reducing the effort of developing high-concurrent applications.**Traditional threads, also called platform threads, are thin wrappers around OS threads. One of the major issues with platform threads is that they run the code on the OS thread and capture the OS thread throughout its lifetime. There is a limit to the number of OS threads, and this creates a scalability bottleneck.

**Like platform threads, a virtual thread is also an instance of *java.lang.Thread* class, but it isn’t tied to a specific OS thread**. It runs the code on a specific OS thread but does not capture the thread for an entire lifetime. Therefore, many virtual threads can share OS threads to run their code.

Let us see the use of the virtual thread with an example:

try(var executor = Executors.newVirtualThreadPerTaskExecutor()) {  
 IntStream.rangeClosed(1, 10\_00).forEach(i -> {  
 executor.submit(() -> {  
 System.out.println(i);  
 try {  
 Thread.sleep(Duration.ofSeconds(1));  
 }  
 catch (InterruptedException e) {  
 e.printStackTrace();  
 }  
 });  
 });  
}

Terminologies 🡪

Boilerplate Code - Sections of code repeated in multiple places with little to no variation.

Data carrier classes - In Java, data carrier classes, often referred to as Data Transfer Objects (DTOs) or Plain Old Java Objects (POJOs), are used to encapsulate data and transfer it between different layers of an application. Here are some key points about these classes:

* POJO: Requires explicit definition of constructors, getters, and setters.
* DTO: Similar to POJO but specifically used for data transfer.
* Record: Provides a concise syntax and is immutable by default.