“Modern” Java

“Modern” Java versions

- Selection of “modern” additions between Java 8 – Java 17
- Optionally: preview of Java 21 (LTS)

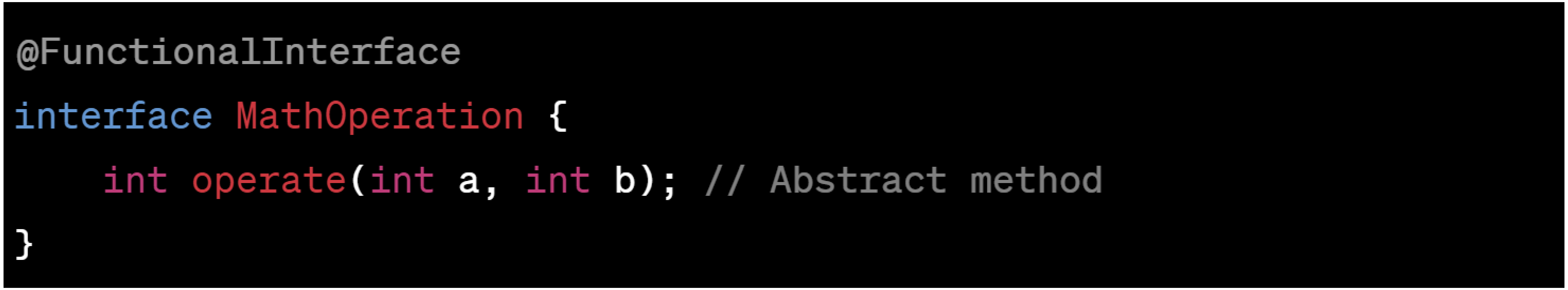
Java 8	• Functional interface; Lambdas • Streams • Optional	8:30 – 12:00ish
Java 10	• var	
Java 14	• Switch expressions	13:00ish – 15:00ish
Java 16	• Pattern matching for instanceof • Records	
Java 17	• Sealed classes	
Java 21	• Sneak peek	15:00ish – 16:30ish

Java 8 - Lambdas

- concise way to represent an anonymous function
- ->
- E.g. *a -> foo(a)*
- Often used to inline implement a *functional interface*

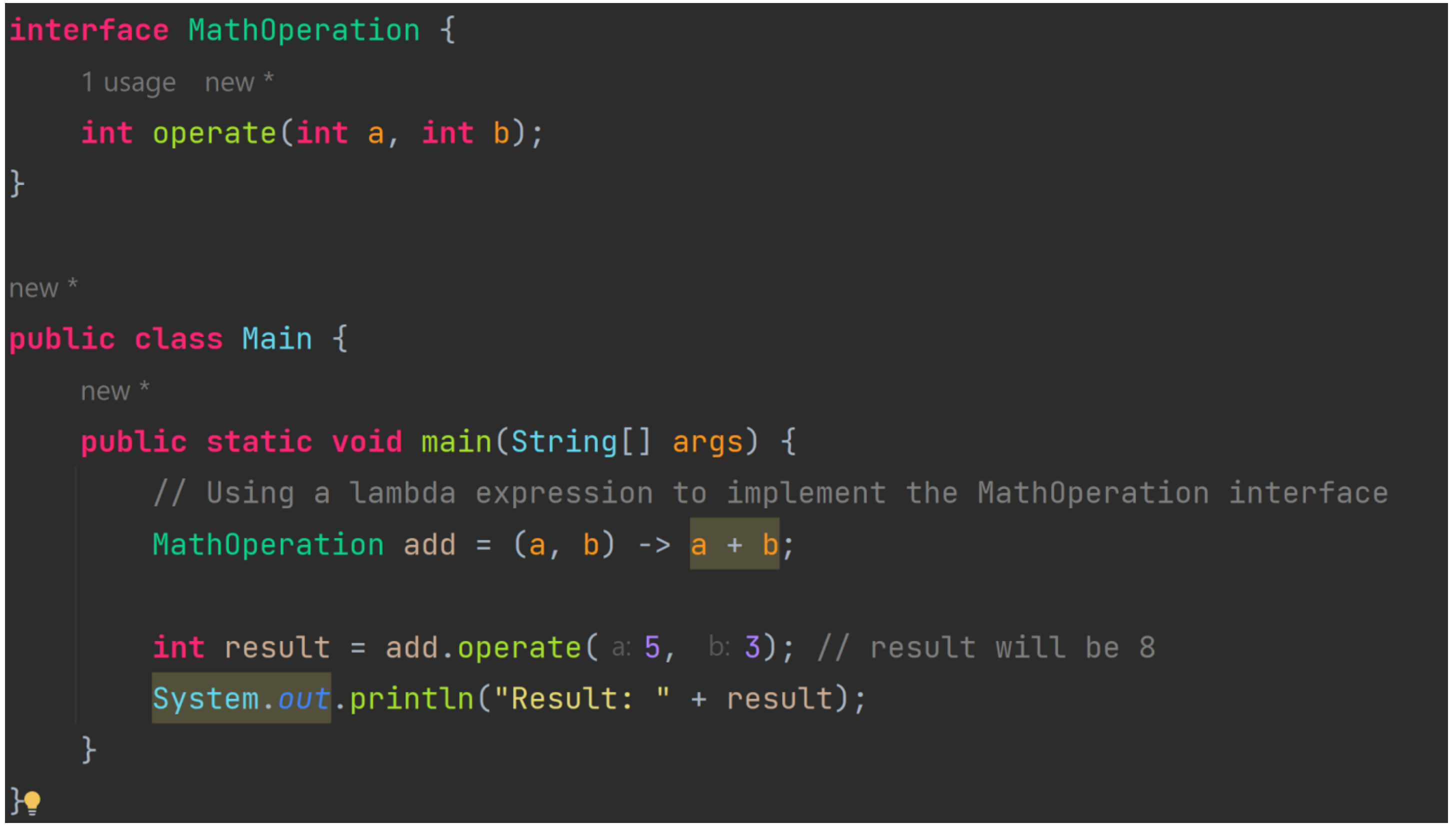
Java 8 – Functional interface

- Interface with exactly one abstract method
- Optionally annotated with *@FunctionalInterface*



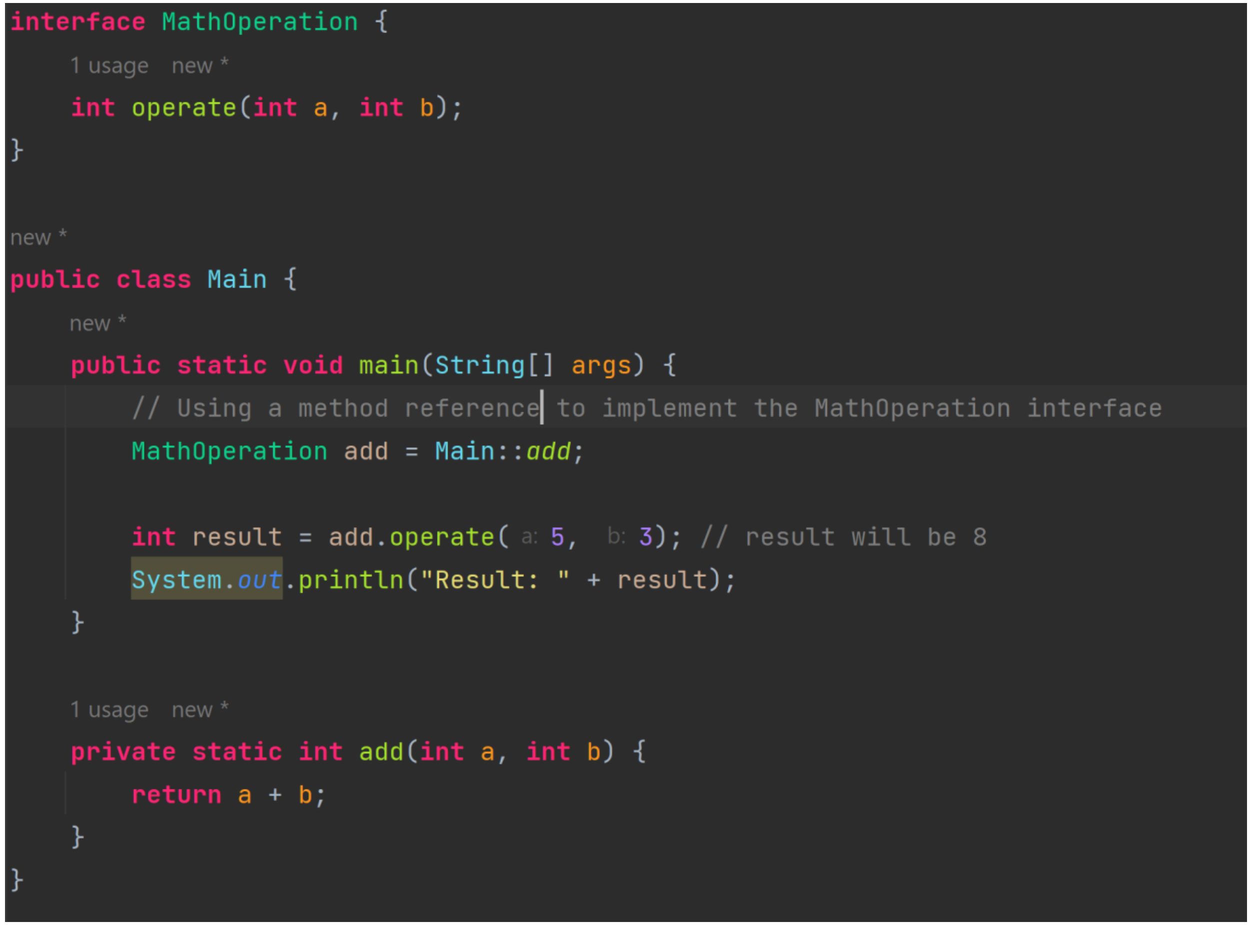
```
@FunctionalInterface
interface MathOperation {
    int operate(int a, int b); // Abstract method
}
```

Java 8 – Lambda expression to implement functional interface



```
interface MathOperation {
    1 usage   new *
    int operate(int a, int b);
}

new *
public class Main {
    new *
    public static void main(String[] args) {
        // Using a lambda expression to implement the MathOperation interface
        MathOperation add = (a, b) -> a + b;

        int result = add.operate( a: 5,  b: 3); // result will be 8
        System.out.println("Result: " + result);
    }
}
```

Java 8 – Method reference to implement functional interface

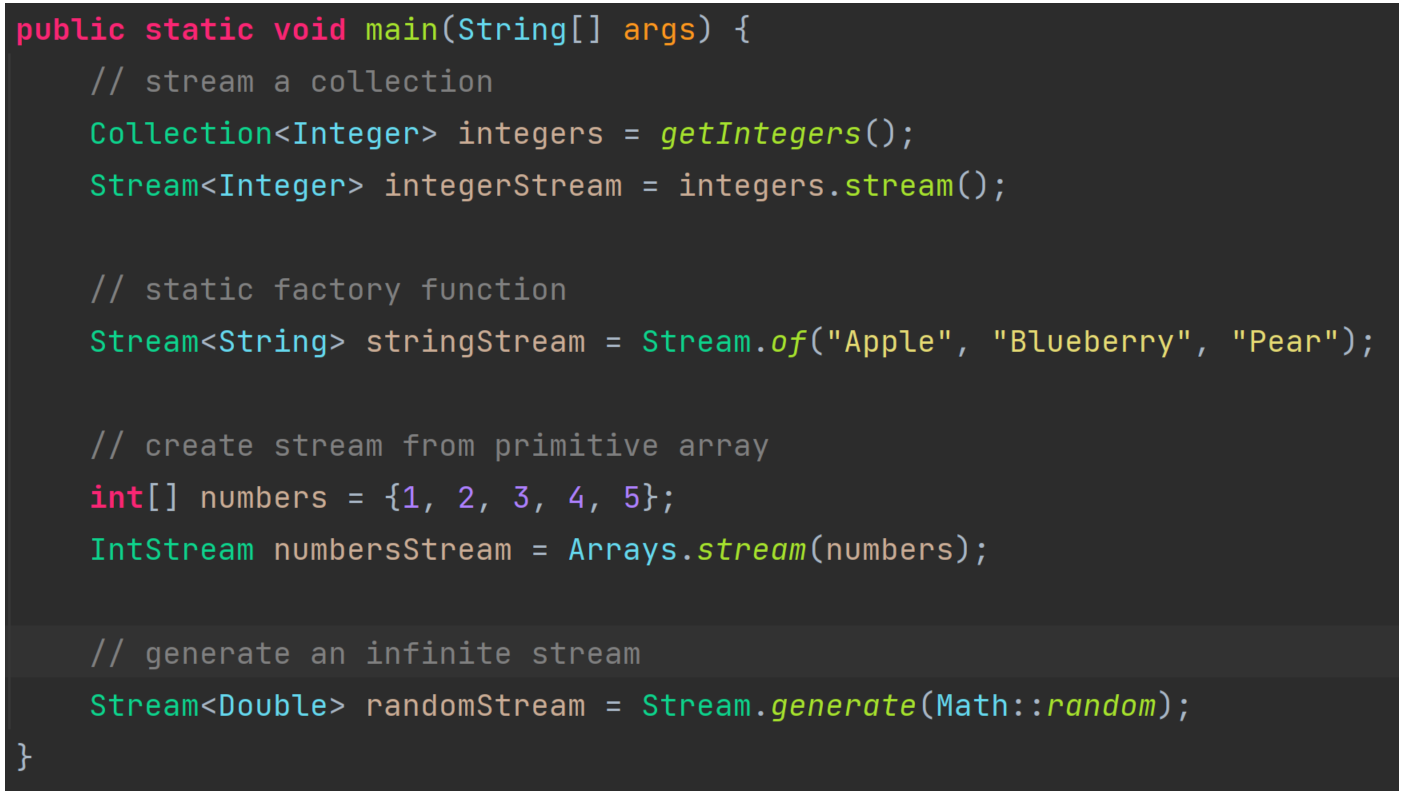


```
interface MathOperation {
    1 usage   new *
    int operate(int a, int b);
}

new *
public class Main {
    new *
    public static void main(String[] args) {
        // Using a method reference to implement the MathOperation interface
        MathOperation add = Main::add;

        int result = add.operate( a: 5,  b: 3); // result will be 8
        System.out.println("Result: " + result);
    }

    1 usage   new *
    private static int add(int a, int b) {
        return a + b;
    }
}
```

Java 8 – Stream API

- Process collections of data in a functional manner
- Concise and expressive code
  - avoiding explicit iteration when dealing with sequences of elements
- Not a datastructure in and of itself
- Streams support various operations that can be performed on the elements, such as **filtering, mapping, sorting, reducing and collecting**.

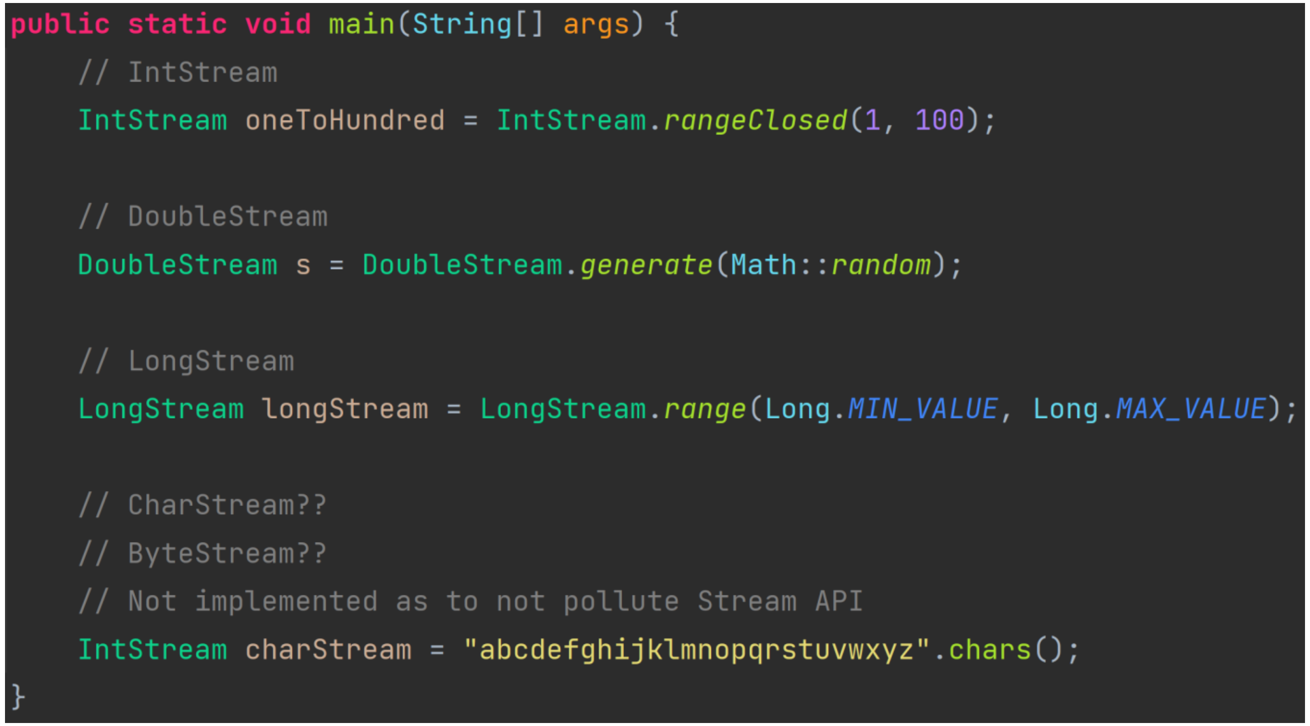
Java 8 – Creating streams



```java
public static void main(String[] args) {
    // stream a collection
    Collection<Integer> integers = getIntegers();
    Stream<Integer> integerStream = integers.stream();

    // static factory function
    Stream<String> stringStream = Stream.of("Apple", "Blueberry", "Pear");

    // create stream from primitive array
    int[] numbers = {1, 2, 3, 4, 5};
    IntStream numbersStream = Arrays.stream(numbers);

    // generate an infinite stream
    Stream<Double> randomStream = Stream.generate(Math::random);
}
```

Java 8 – Primitive streams

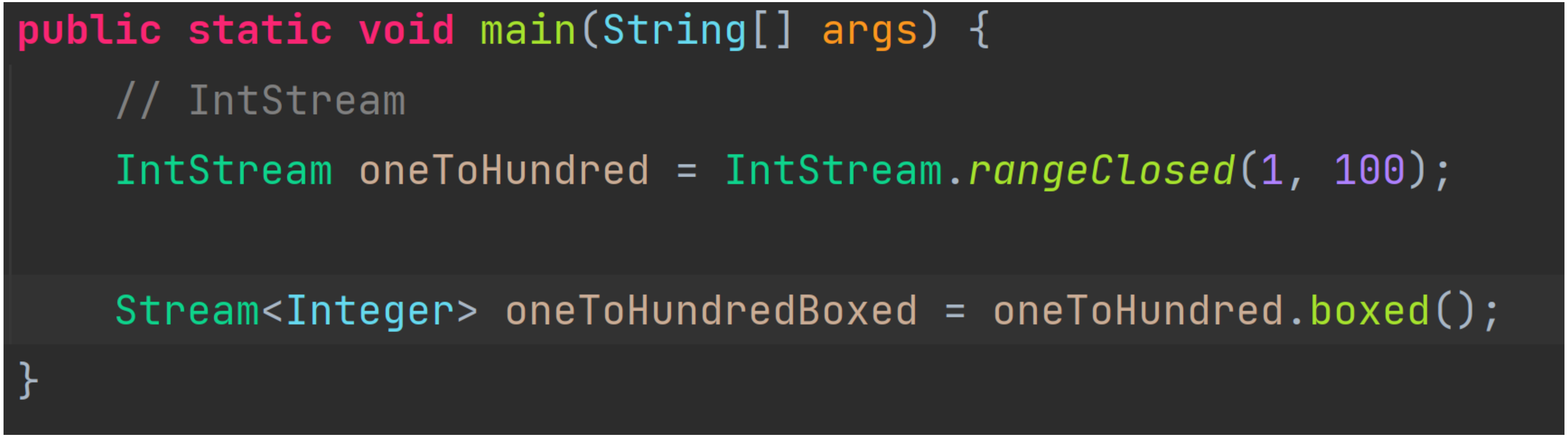
- Streams work primarily on objects
- I want to create a stream of primitive type? 2 options:
  - Use boxed types
  - Use primitive streams



```java
public static void main(String[] args) {
    // IntStream
    IntStream oneToHundred = IntStream.rangeClosed(1, 100);

    // DoubleStream
    DoubleStream s = DoubleStream.generate(Math::random);

    // LongStream
    LongStream longStream = LongStream.range(Long.MIN_VALUE, Long.MAX_VALUE);

    // CharStream??
    // ByteStream??
    // Not implemented as to not pollute Stream API
    IntStream charStream = "abcdefghijklmnopqrstuvwxyz".chars();
}
```

Java 8 – Converting primitive to boxed Stream

- .boxed()



```
public static void main(String[] args) {
    // IntStream
    IntStream oneToHundred = IntStream.rangeClosed(1, 100);

    Stream<Integer> oneToHundredBoxed = oneToHundred.boxed();
}
```

Java 8 – Stream operations

- .filter
- .map
- .forEach
- .reduce
- .collect
- .findAny / .findFirst
- …

Java 8 – Stream operations

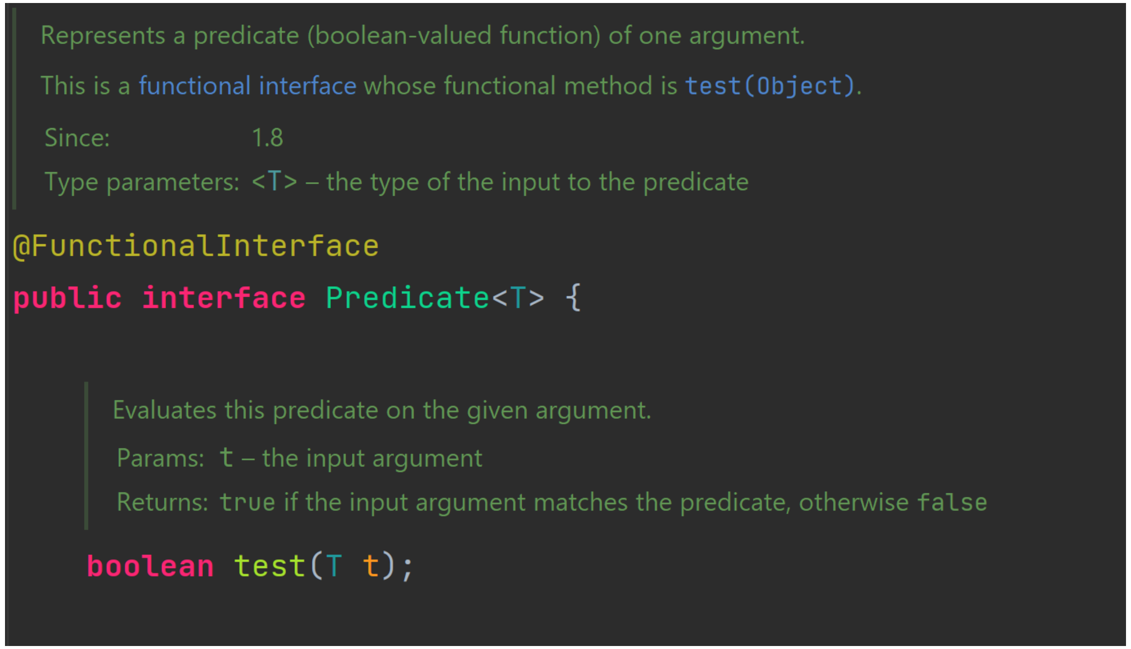
- .filter
- .map/.flatMap
- .forEach
- .reduce
- .collect
- .findAny / .findFirst
- …

accepts

- Predicate
- Function
- Consumer
- BinaryOperator
- Collector
- No args
- …

Functional interfaces

Java 8 – filtering



Represents a predicate (boolean-valued function) of one argument.

This is a functional interface whose functional method is `test(Object)`.

Since: 1.8

Type parameters: <T> – the type of the input to the predicate

```
@FunctionalInterface
public interface Predicate<T> {
```

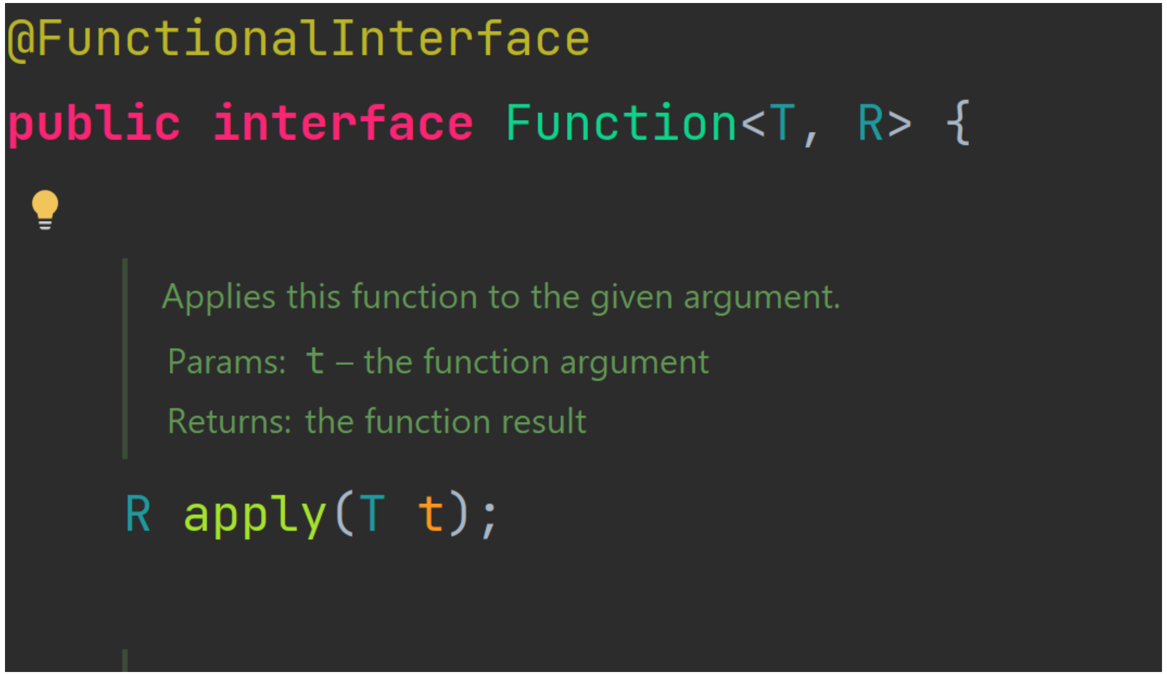
Evaluates this predicate on the given argument.

Params: `t` – the input argument

Returns: `true` if the input argument matches the predicate, otherwise `false`

```
    boolean test(T t);
```

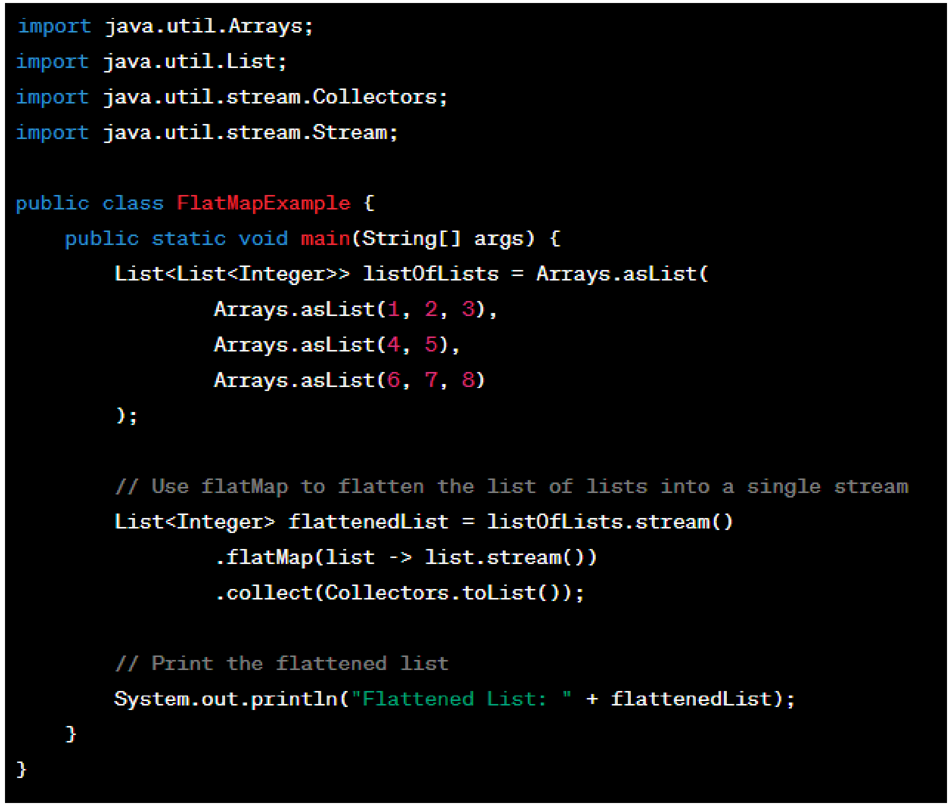
Java 8 – mapping



```
@FunctionalInterface
public interface Function<T, R> {

    Applies this function to the given argument.
    Params: t – the function argument
    Returns: the function result
    R apply(T t);
```

Java 8 – flatMap

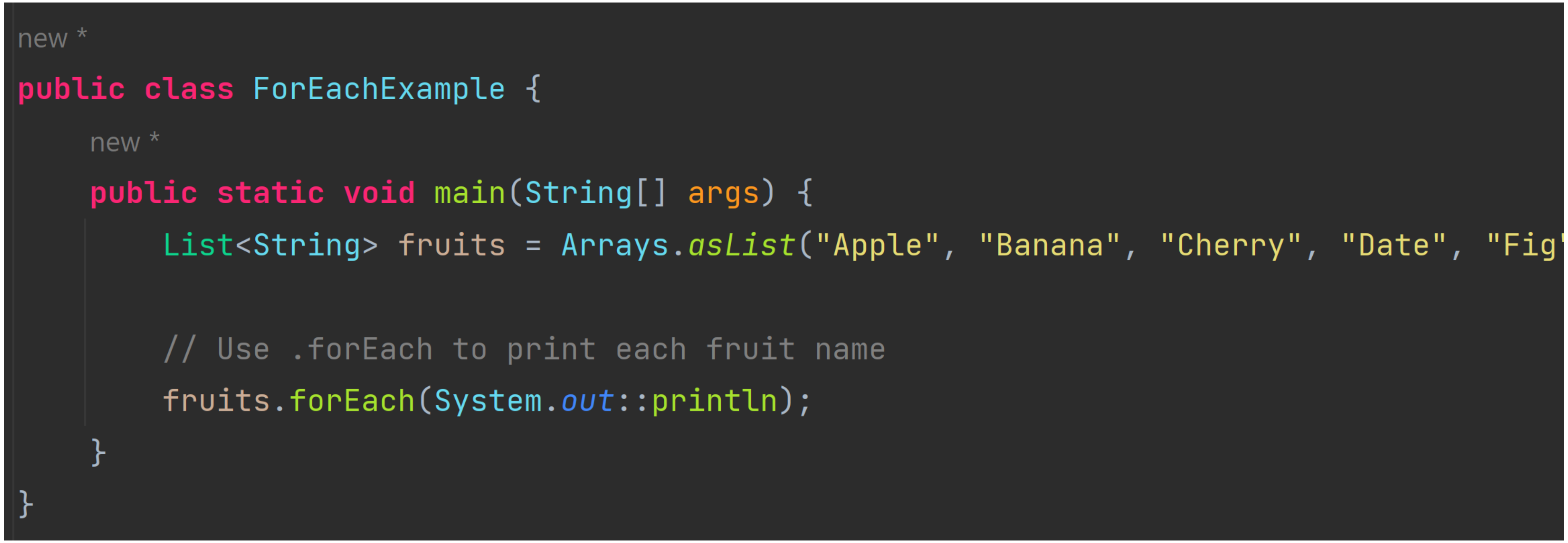
- Two step operation:
  1. Transform element into stream
  2. Flatten resulting streams



```java
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;
import java.util.stream.Stream;

public class FlatMapExample {
    public static void main(String[] args) {
        List<List<Integer>> listOfLists = Arrays.asList(
                Arrays.asList(1, 2, 3),
                Arrays.asList(4, 5),
                Arrays.asList(6, 7, 8)
        );

        // Use flatMap to flatten the list of lists into a single stream
        List<Integer> flattenedList = listOfLists.stream()
                .flatMap(list -> list.stream())
                .collect(Collectors.toList());

        // Print the flattened list
        System.out.println("Flattened List: " + flattenedList);
    }
}
```

Java 8 – forEach

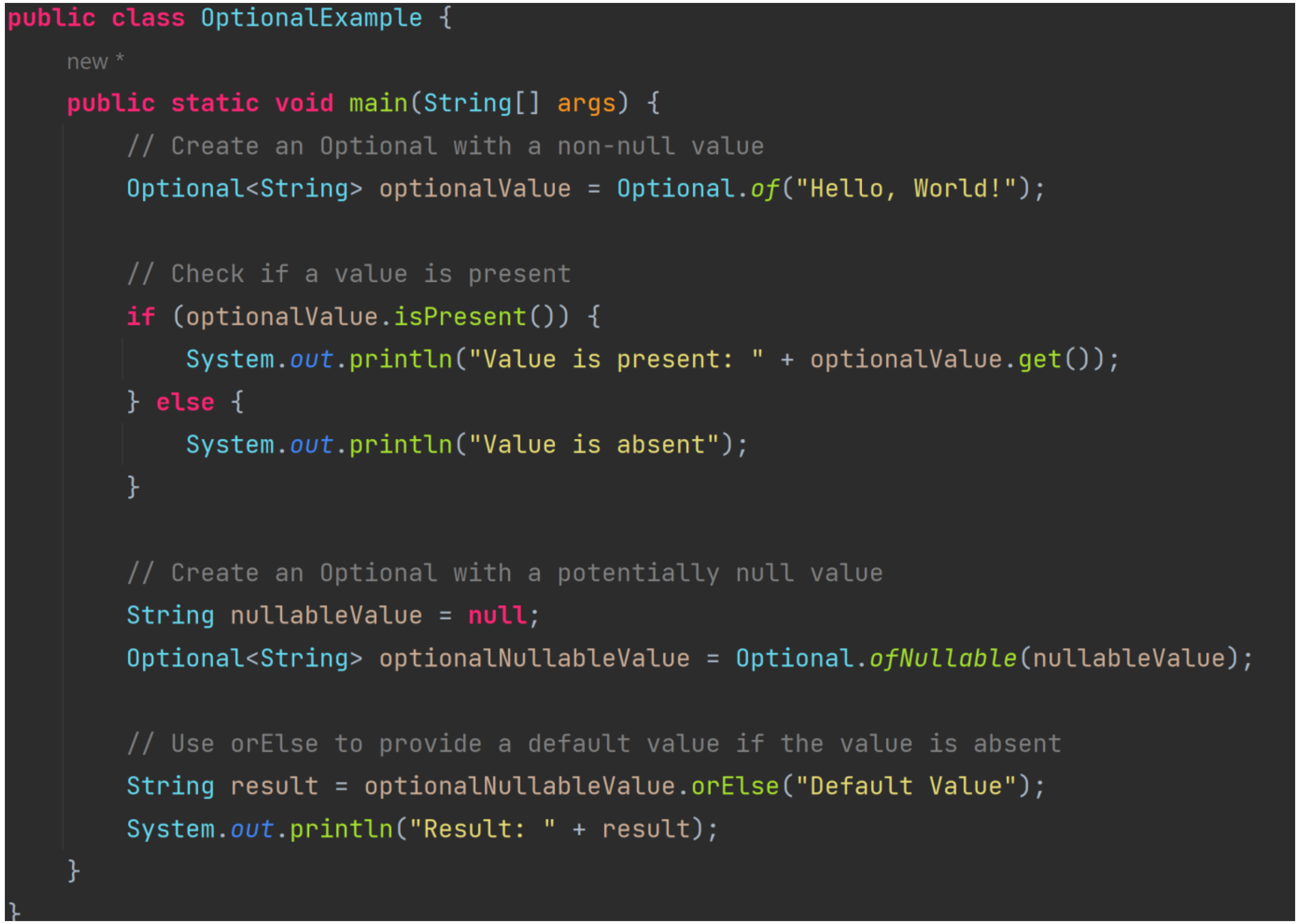
- Process elements



```
new *
public class ForEachExample {
    new *
    public static void main(String[] args) {
        List<String> fruits = Arrays.asList("Apple", "Banana", "Cherry", "Date", "Fig

        // Use .forEach to print each fruit name
        fruits.forEach(System.out::println);
    }
}
```

Java 8 – Optional<T>

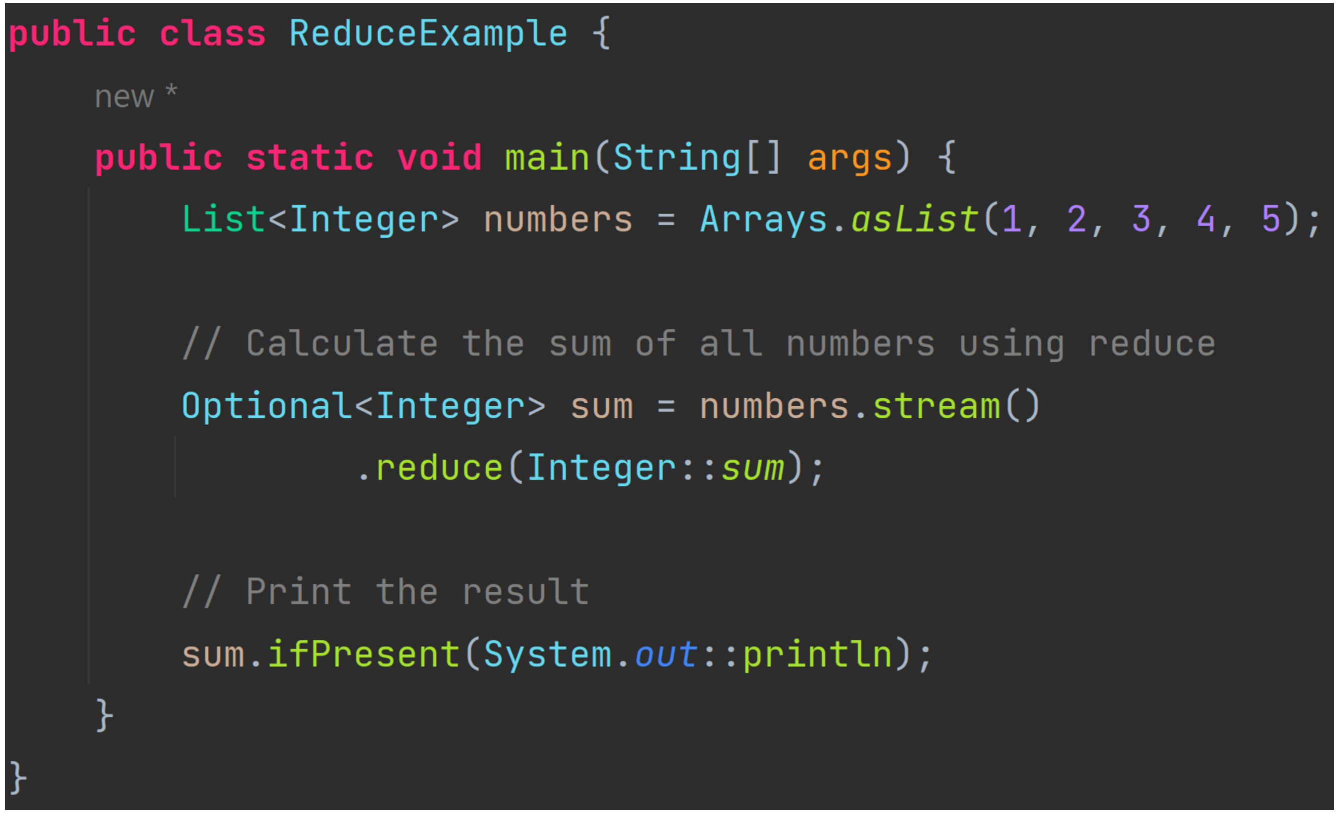
- May or may not contain a non-null value of type T
- Useful in situations where a “solution” might not be possible
- Avoid NPE
- .isPresent, .ifPresent, .orElse, .orElseThrow…



```java
public class OptionalExample {
    public static void main(String[] args) {
        // Create an Optional with a non-null value
        Optional<String> optionalValue = Optional.of("Hello, World!");

        // Check if a value is present
        if (optionalValue.isPresent()) {
            System.out.println("Value is present: " + optionalValue.get());
        } else {
            System.out.println("Value is absent");
        }

        // Create an Optional with a potentially null value
        String nullableValue = null;
        Optional<String> optionalNullableValue = Optional.ofNullable(nullableValue);

        // Use orElse to provide a default value if the value is absent
        String result = optionalNullableValue.orElse("Default Value");
        System.out.println("Result: " + result);
    }
}
```

Java 8 – reduce

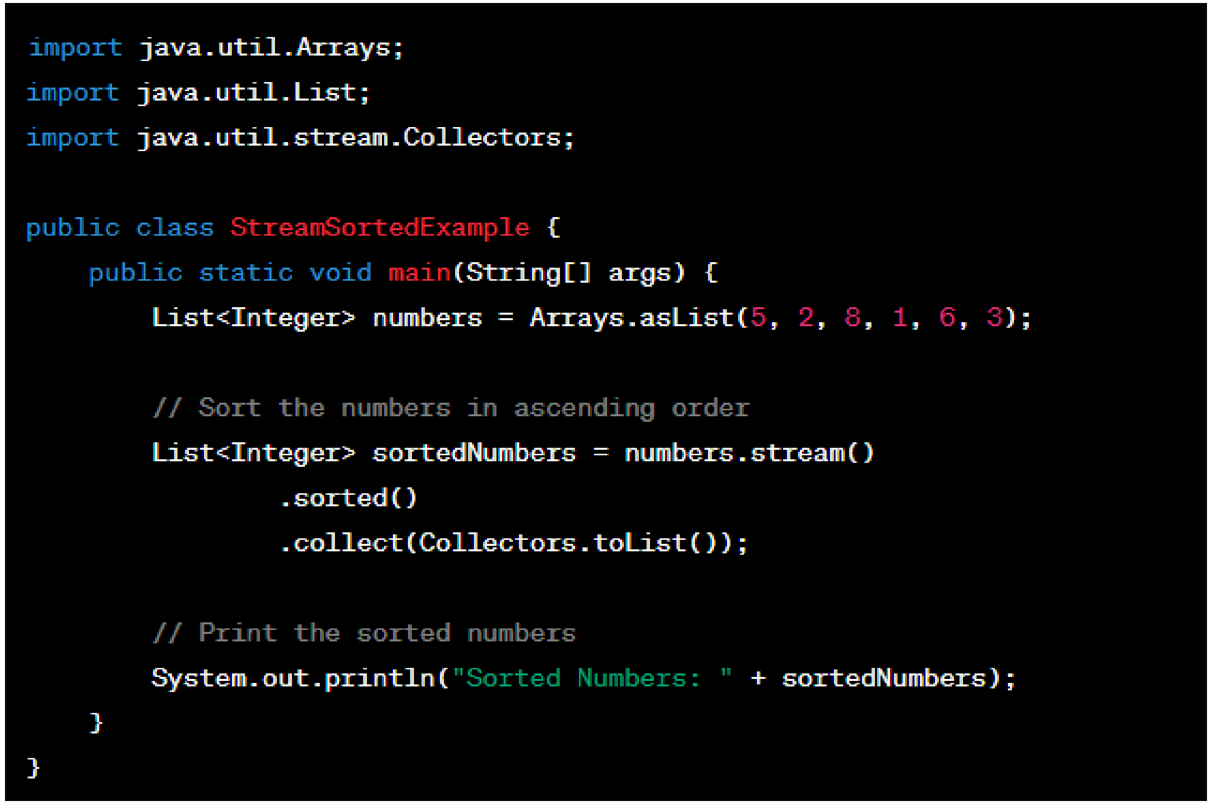
- Reduce elements of a stream into a single element using an implementation of the BinaryOperator functional interface
- Returns an Optional → why?



```
public class ReduceExample {
    new *
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

        // Calculate the sum of all numbers using reduce
        Optional<Integer> sum = numbers.stream()
                .reduce(Integer::sum);

        // Print the result
        sum.ifPresent(System.out::println);
    }
}
```

Java 8 – sorted

- Sort elements based on comparator logic represented by implementation of functional Comparator interface



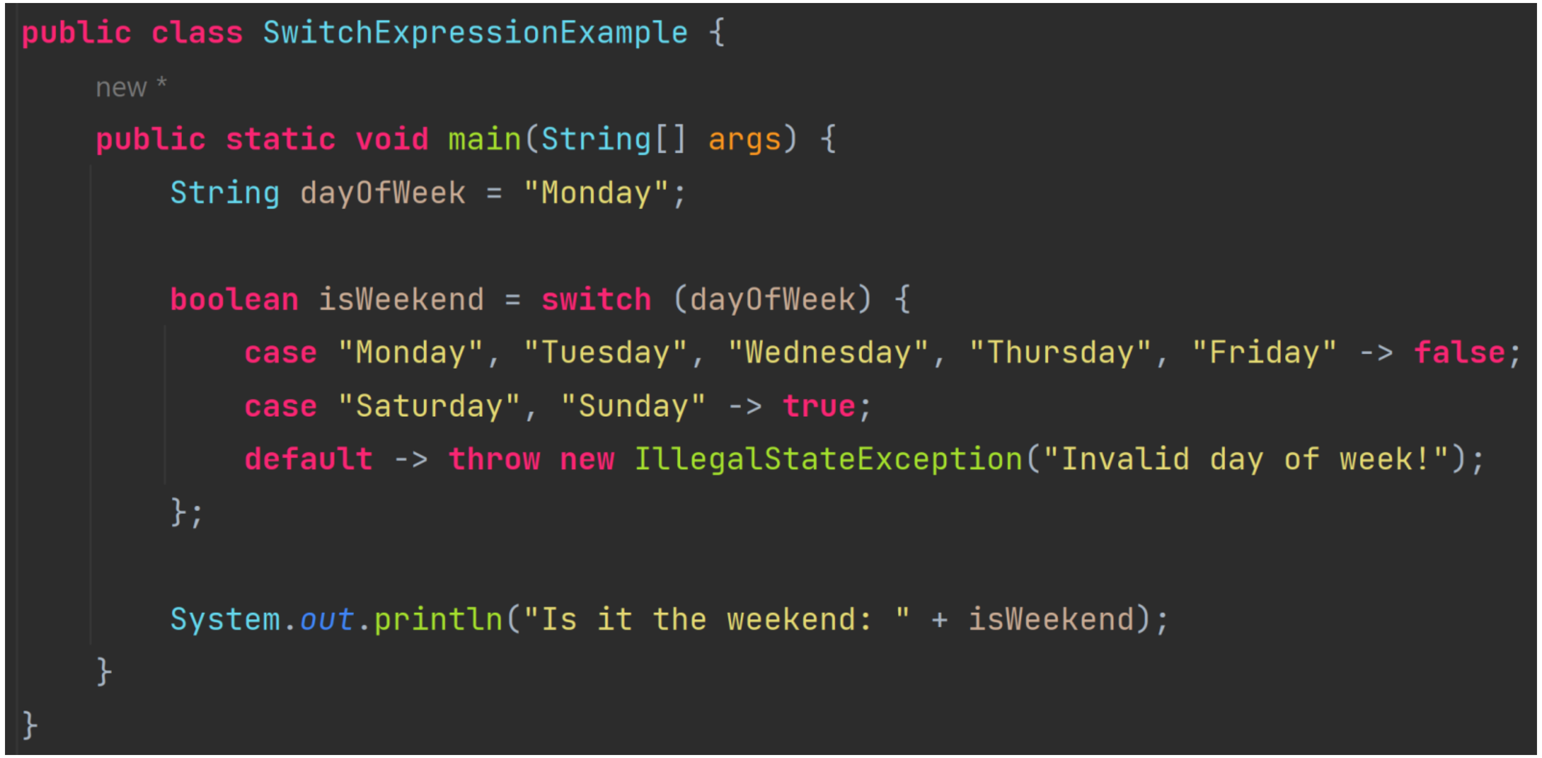
```
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class StreamSortedExample {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(5, 2, 8, 1, 6, 3);

        // Sort the numbers in ascending order
        List<Integer> sortedNumbers = numbers.stream()
                .sorted()
                .collect(Collectors.toList());

        // Print the sorted numbers
        System.out.println("Sorted Numbers: " + sortedNumbers);
    }
}
```

Java 10 – var

- Local variable type inference
- Project Amber: improve code readability and boilerplate code
- Use with caution!!
    - Okay e.g. when looping over the entry set of a map where the key and value types are clear
    - Okay when prototyping something like a stream
    - Not okay to use when lazy

Java 14 – switch expression

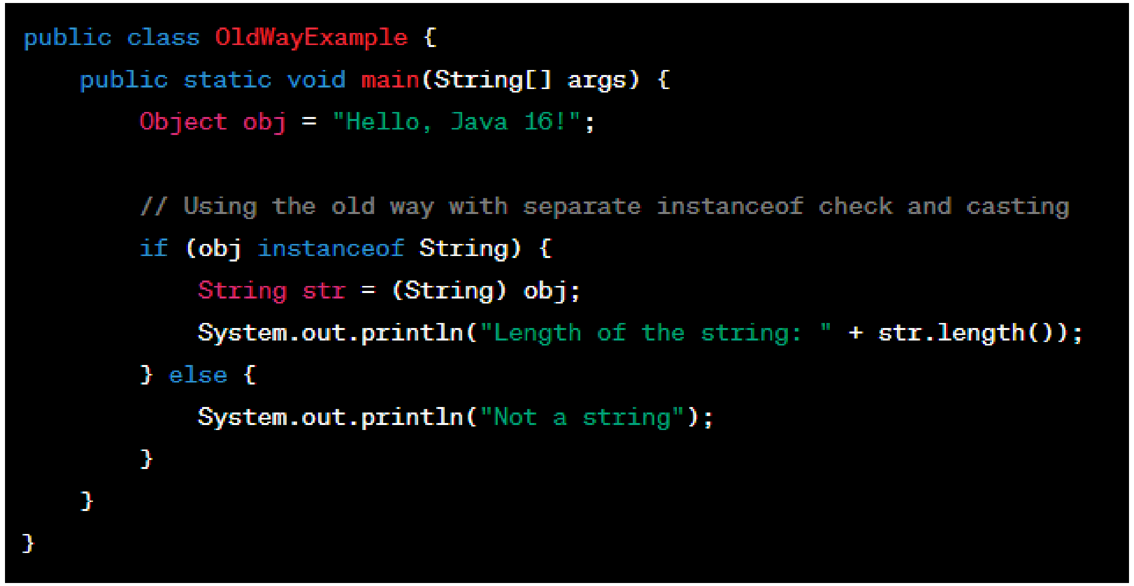
- Concise way to handle branching logic in a single expression
- The result of the matched case is returned as the result of the expression
- -> and -> {yield}



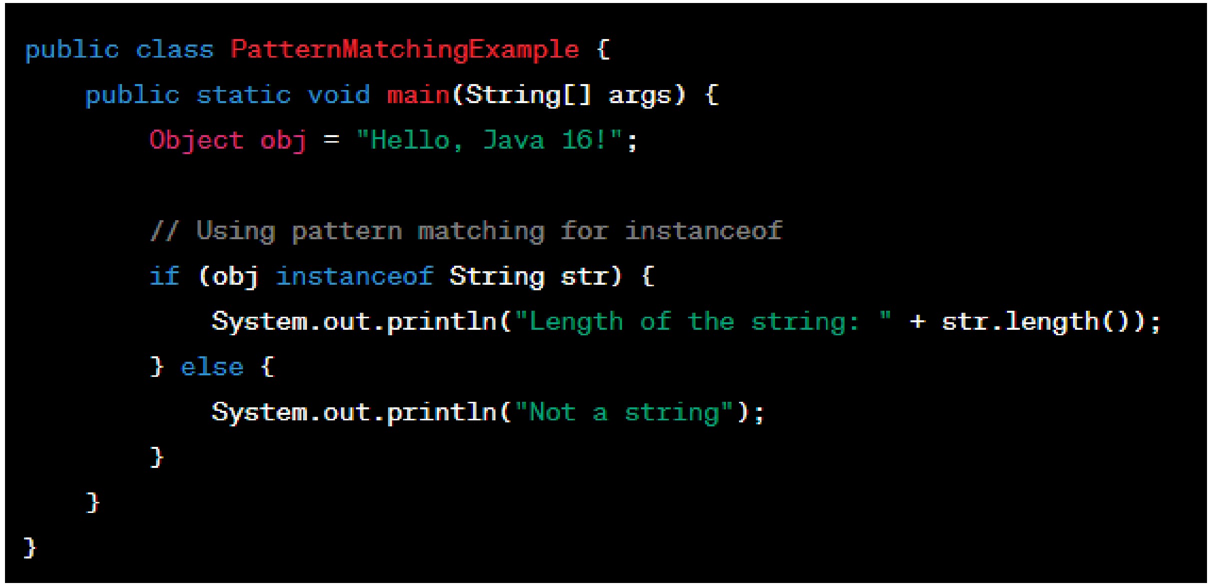
```
public class SwitchExpressionExample {
    new *
    public static void main(String[] args) {
        String dayOfWeek = "Monday";

        boolean isWeekend = switch (dayOfWeek) {
            case "Monday", "Tuesday", "Wednesday", "Thursday", "Friday" -> false;
            case "Saturday", "Sunday" -> true;
            default -> throw new IllegalStateException("Invalid day of week!");
        };

        System.out.println("Is it the weekend: " + isWeekend);
    }
}
```

Java 16 – Pattern matching for instanceof

- Simplifies pattern of checking type with instanceof and thereafter casting it



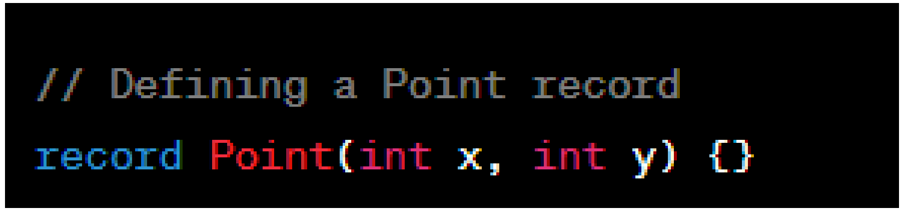
```java
public class OldWayExample {
    public static void main(String[] args) {
        Object obj = "Hello, Java 16!";

        // Using the old way with separate instanceof check and casting
        if (obj instanceof String) {
            String str = (String) obj;
            System.out.println("Length of the string: " + str.length());
        } else {
            System.out.println("Not a string");
        }
    }
}
```



```java
public class PatternMatchingExample {
    public static void main(String[] args) {
        Object obj = "Hello, Java 16!";

        // Using pattern matching for instanceof
        if (obj instanceof String str) {
            System.out.println("Length of the string: " + str.length());
        } else {
            System.out.println("Not a string");
        }
    }
}
```

Java 16 – Records

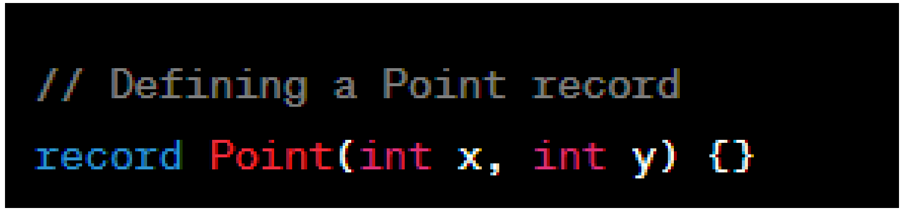
- Reduce boilerplate
  - ~~Getters/Setters~~
  - ~~.equals~~
  - ~~.hashcode~~
  - ~~.toString~~
- Automatic getters
- Can have member methods just like a class
- Useful for DTOs and data-centric programming



```
// Defining a Point record
record Point(int x, int y) {}
```

Java 17 – Sealed classes

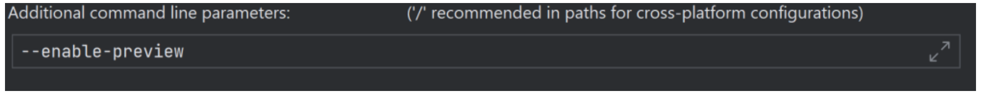
- Specify a limited set of allowed subclassed → ‘permits’ modifier
- Disallow any other inheritance → ‘sealed’ modifier
- Make inheritance hierarchy more predictable and maintainable
- **Solves instanceof checking smells**
- Very useful for large frameworks/libraries to constrain inheritance in large class hierarchies



```java
// Defining a Point record
record Point(int x, int y) {}
```

Java 21

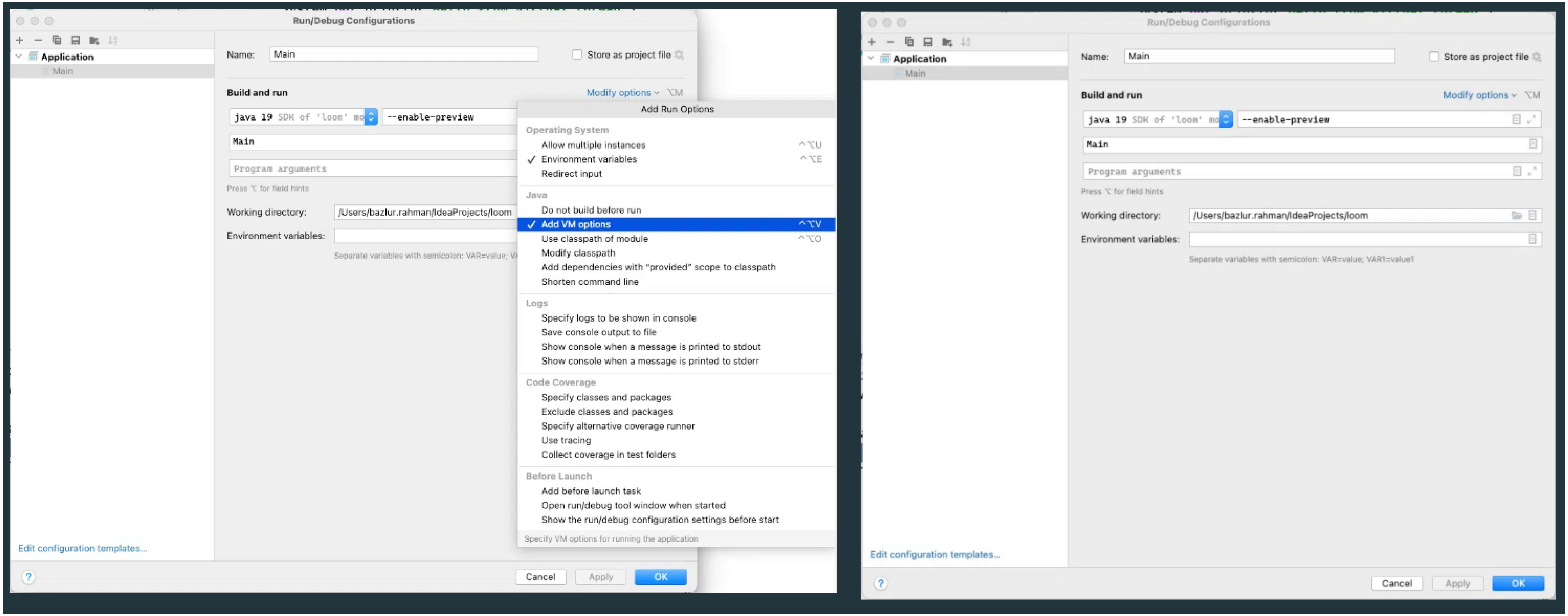
- Release: tomorrow 19/09
- We will be using latest preview build: https://jdk.java.net/21/
- Some set-up in IntelliJ:
  - Project Structure > language level – X
  - Project Settings > Java Compiler >



Additional command line parameters: ('/' recommended in paths for cross-platform configurations)

```
--enable-preview
```

  - Run configuration > (see next slide)



Run/Debug Configurations

Name: Main

□ Store as project file

Build and run

java 19 SDK of 'loom' mo

--enable-preview

Main

Program arguments

Press ⌥ for field hints

Working directory: /Users/bazlur.rahman/IdeaProjects/loom

Environment variables:

Separate variables with semicolon: VAR=value; V

Run/Debug Configurations

Name: Main

□ Store as project file

Build and run

java 19 SDK of 'loom' mo

--enable-preview

Main

Program arguments

Press ⌥ for field hints

Working directory: /Users/bazlur.rahman/IdeaProjects/loom

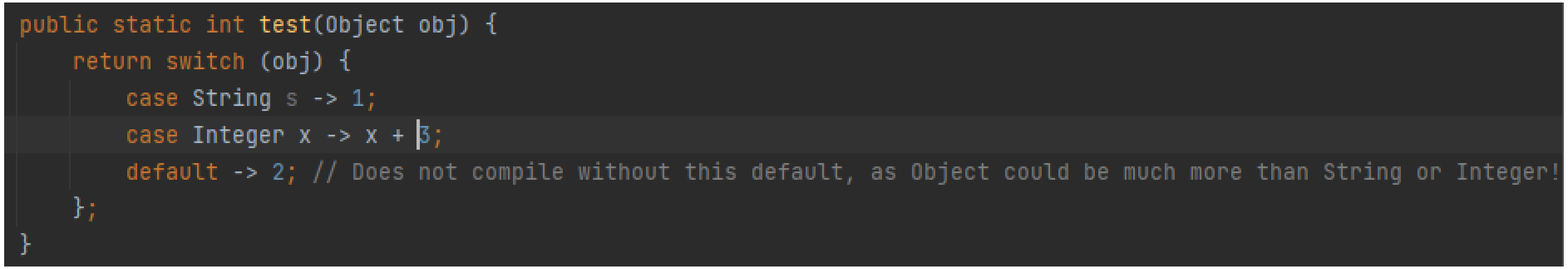
Environment variables:

Separate variables with semicolon: VAR=value; VAR1=value1

Application
Main
Modify options ⌥M
Add Run Options
Operating System
Allow multiple instances ^⌥U
✓ Environment variables ^⌥E
Redirect input
Java
Do not build before run
✓ Add VM options ^⌥V
Use classpath of module ^⌥O
Modify classpath
Add dependencies with "provided" scope to classpath
Shorten command line
Logs
Specify logs to be shown in console
Save console output to file
Show console when a message is printed to stdout
Show console when a message is printed to stderr
Code Coverage
Specify classes and packages
Exclude classes and packages
Specify alternative coverage runner
Use tracing
Collect coverage in test folders
Before Launch
Add before launch task
Open run/debug tool window when started
Show the run/debug configuration settings before start
Specify VM options for running the application
Edit configuration templates...
Cancel
Apply
OK
Application
Main
Modify options ⌥M
Edit configuration templates...
Cancel
Apply
OK

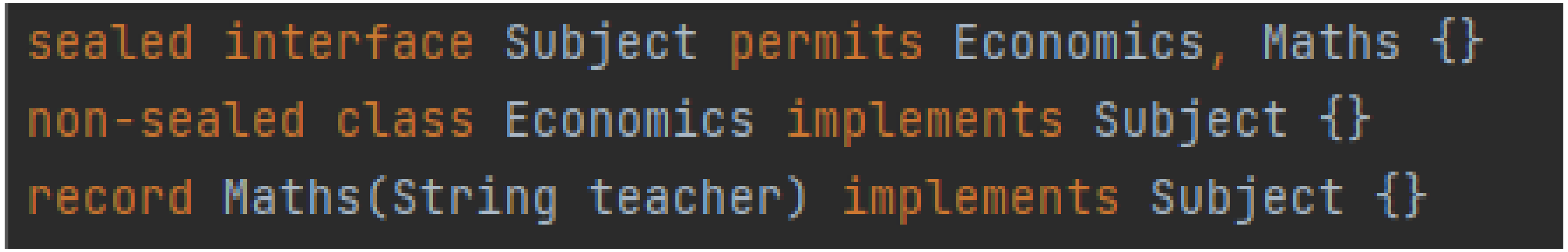
Java 21 – Pattern Matching for Switch

- Avoid huge if – else if – else statements when type checking
  - Switch statements!
  - Before 21: only Strings, enum’s and (boxed) integer primitives
  - Now every type
  - Auto exhaustiveness checks for pattern-matched switch statements! (not for normal switch statemens to keep backwards compatibility)

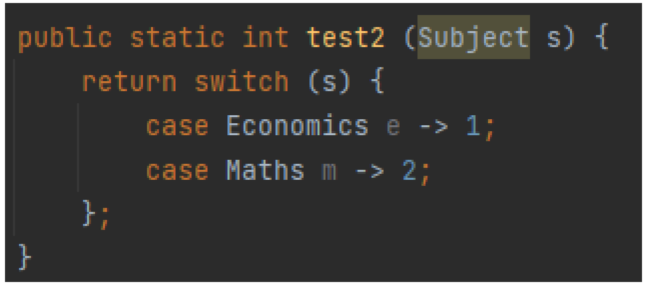


```java
public static int test(Object obj) {
    return switch (obj) {
        case String s -> 1;
        case Integer x -> x + 3;
        default -> 2; // Does not compile without this default, as Object could be much more than String or Integer!
    };
}
```

Java 21 – Pattern Matching for Switch



```
sealed interface Subject permits Economics, Maths {}
non-sealed class Economics implements Subject {}
record Maths(String teacher) implements Subject {}
```

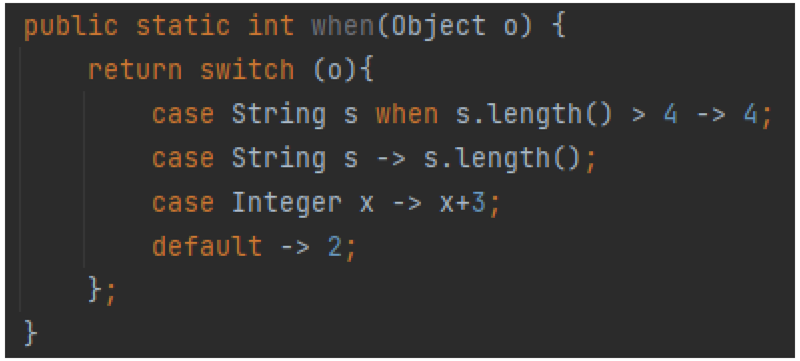


```
public static int test2 (Subject s) {
    return switch (s) {
        case Economics e -> 1;
        case Maths m -> 2;
    };
}
```

Sealed classes => This switch is exhaustive

Java 21 – Pattern Matching for Switch

- Guards verminderen de hoeveelheid conditional code binnen een case: keyword *when* gevolgd door booleaanse expressive
- The first case match is followed



```
public static int when(Object o) {
    return switch (o){
        case String s when s.length() > 4 -> 4;
        case String s -> s.length();
        case Integer x -> x+3;
        default -> 2;
    };
}
```

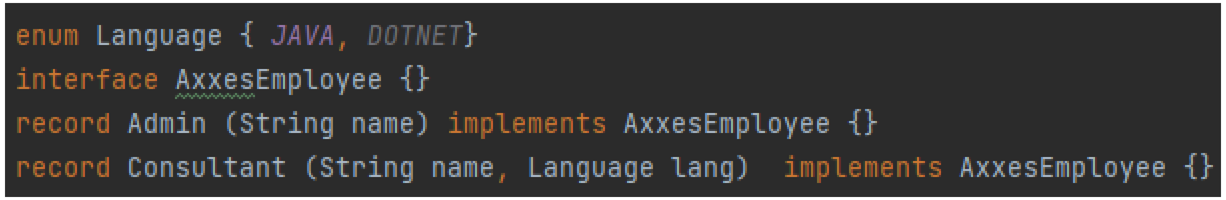
Java 21 – Pattern Matching for Switch

Some *minor* changes

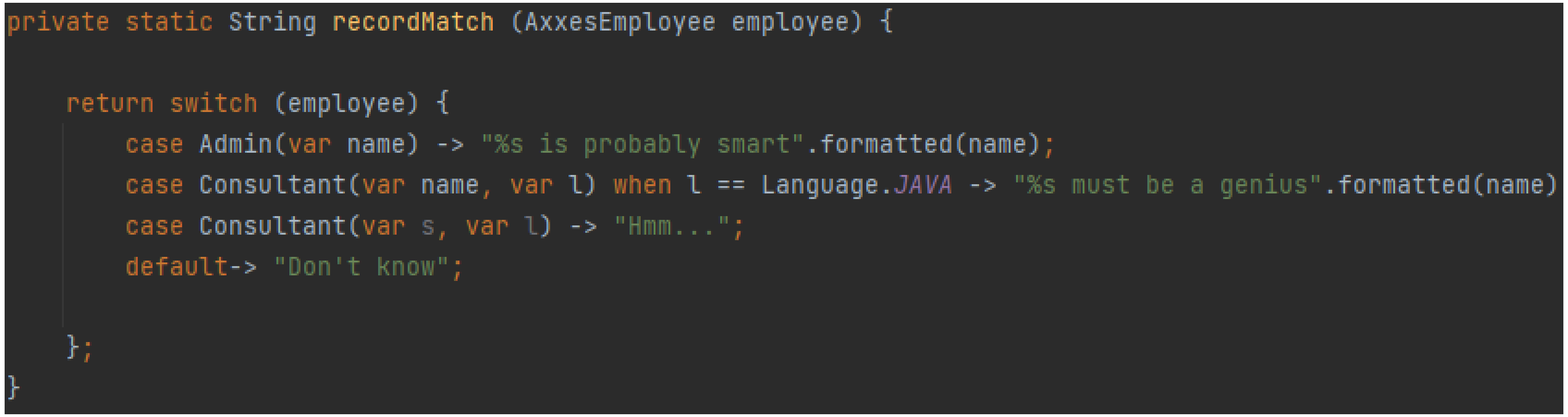
- Null checks can be inside the switch (NPE before J21)
- Pattern matching cases do not allow fall-through!

Java 21 – Record patterns

- A new *construct* is now allowed when doing instanceof on a record or in a switch case
- RecordPattern = *RecordType* (…fields of record)
- Most value when having nested records

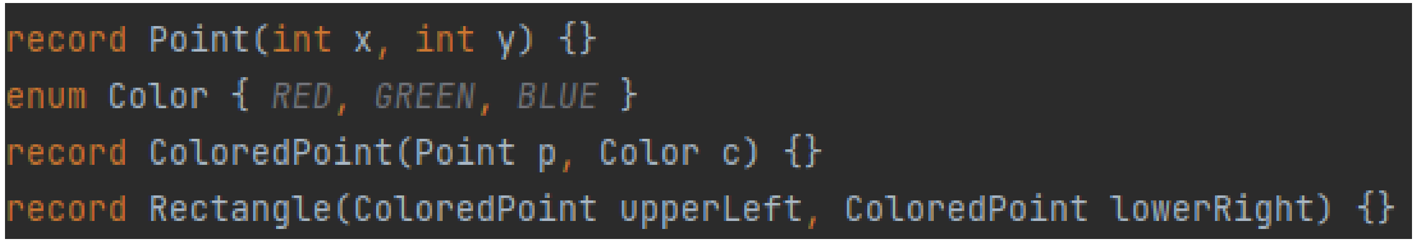


```
enum Language { JAVA, DOTNET}
interface AxxesEmployee {}
record Admin (String name) implements AxxesEmployee {}
record Consultant (String name, Language lang)  implements AxxesEmployee {}
```

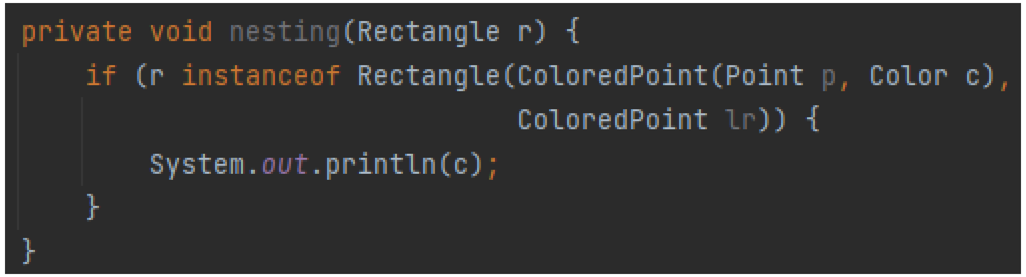


```
private static String recordMatch (AxxesEmployee employee) {

    return switch (employee) {
        case Admin(var name) -> "%s is probably smart".formatted(name);
        case Consultant(var name, var l) when l == Language.JAVA -> "%s must be a genius".formatted(name)
        case Consultant(var s, var l) -> "Hmm...";
        default-> "Don't know";

    };
}
```

Java 21 – Record patterns



```java
record Point(int x, int y) {}
enum Color { RED, GREEN, BLUE }
record ColoredPoint(Point p, Color c) {}
record Rectangle(ColoredPoint upperLeft, ColoredPoint lowerRight) {}
```



```java
private void nesting(Rectangle r) {
    if (r instanceof Rectangle(ColoredPoint(Point p, Color c),
                               ColoredPoint lr)) {
        System.out.println(c);
    }
}
```

Java 21 – Virtual Threads

Pre Java 21: Every Java thread wraps a OS level thread

Thread-per-request applications: every request runs on its own thread

=> very limited throughput!

Virtual Threads

- Decoupling of Java (virtual) Thread & OS threads
- Unlock underlying OS thread (« give it back to the pool ») when operation blocks (i.e. networking)
- « Using virtual threads does not require learning new concepts, it may require *unlearning* habits.»

Java 21 – Virtual Threads

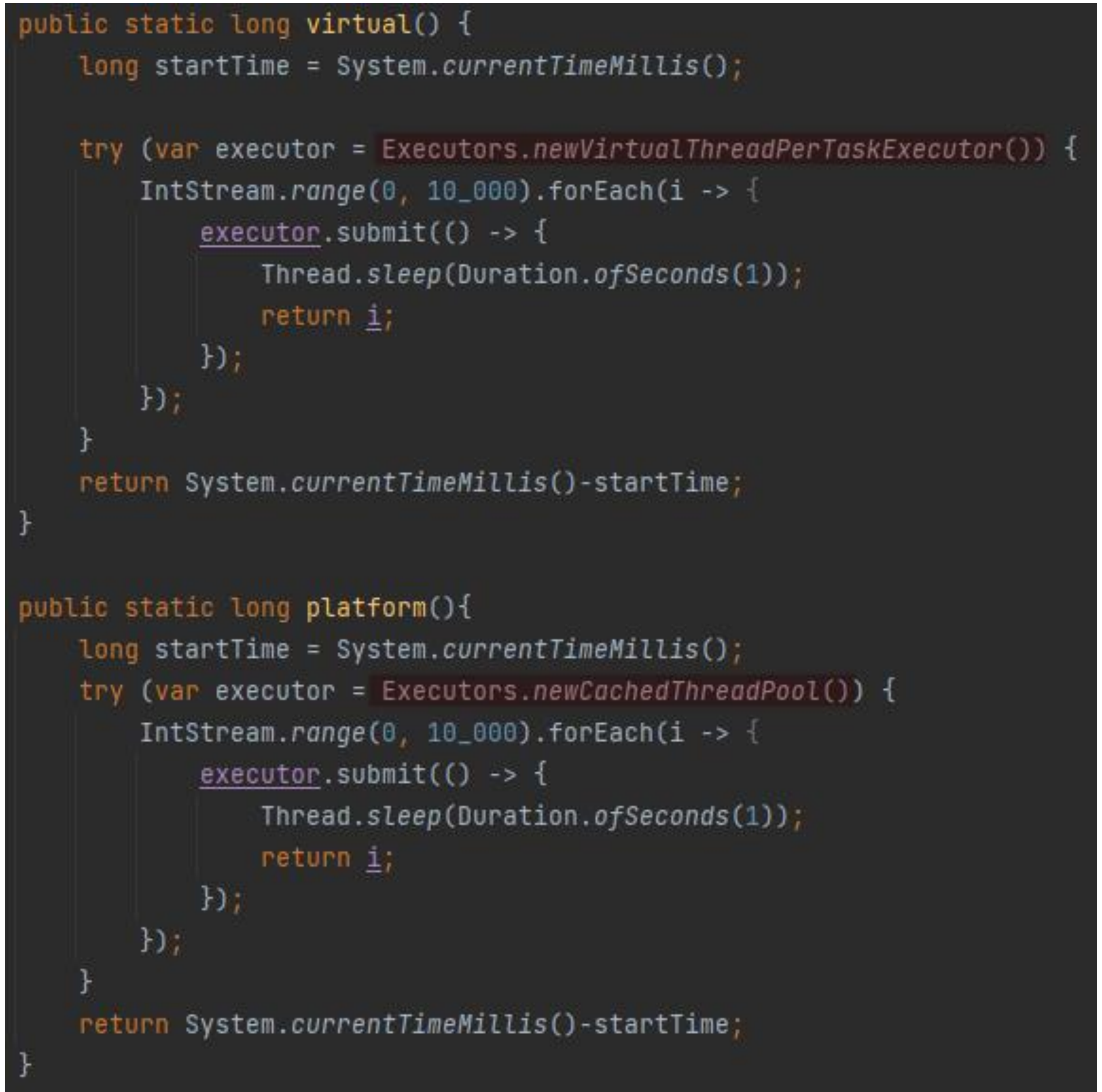
Virtual threads creation:

- Executors.newVirtualThreadPerTaskExecutor()
- Thread.Builder API
- Structured concurrency (Next topic)

Terminology:

- Carrier: Platform thread that (instigated by virtual) runs on OS thread
- (un)mounting: (de)coupling Virtual thread from/to *carrier.*
- Pinned: Virtual Thread that cannot unmount from carrier because
  - It’s in a synchronized block
  - Executing native methods/foreign functions

Java 21 – Virtual Threads



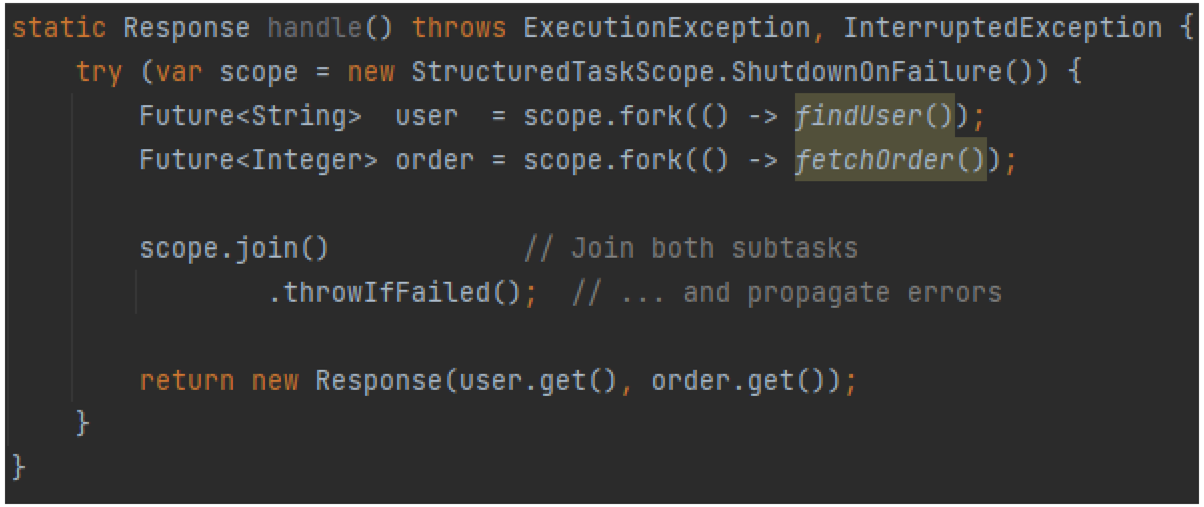
```
public static long virtual() {
    long startTime = System.currentTimeMillis();

    try (var executor = Executors.newVirtualThreadPerTaskExecutor()) {
        IntStream.range(0, 10_000).forEach(i -> {
            executor.submit(() -> {
                Thread.sleep(Duration.ofSeconds(1));
                return i;
            });
        });
    }
    return System.currentTimeMillis()-startTime;
}

public static long platform(){
    long startTime = System.currentTimeMillis();
    try (var executor = Executors.newCachedThreadPool()) {
        IntStream.range(0, 10_000).forEach(i -> {
            executor.submit(() -> {
                Thread.sleep(Duration.ofSeconds(1));
                return i;
            });
        });
    }
    return System.currentTimeMillis()-startTime;
}
```

Java 21 – Virtual Threads

- https://www.youtube.com/watch?v=kirhhcFAGB4

Java 21 – Structured Concurrency (Preview)

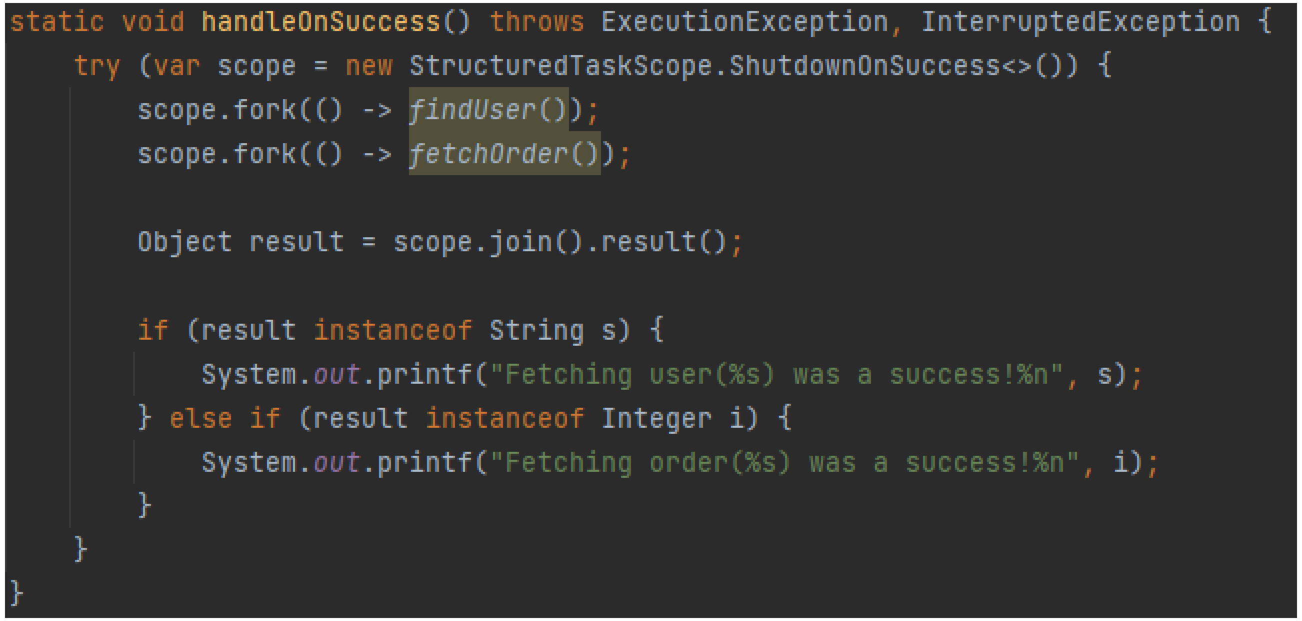
- Helps in dealing with multiple concurrent tasks
- Make the logical relationship between subtaks explicit
- « ShutdownOnFailure »: cancel all when one fails
- « ShutdownOnSuccess » cancel all when first succeeds



```java
static Response handle() throws ExecutionException, InterruptedException {
    try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {
        Future<String>  user  = scope.fork(() -> findUser());
        Future<Integer> order = scope.fork(() -> fetchOrder());

        scope.join()            // Join both subtasks
                .throwIfFailed();  // ... and propagate errors

        return new Response(user.get(), order.get());
    }
}
```

Java 21 – Structured Concurrency (**Preview**)

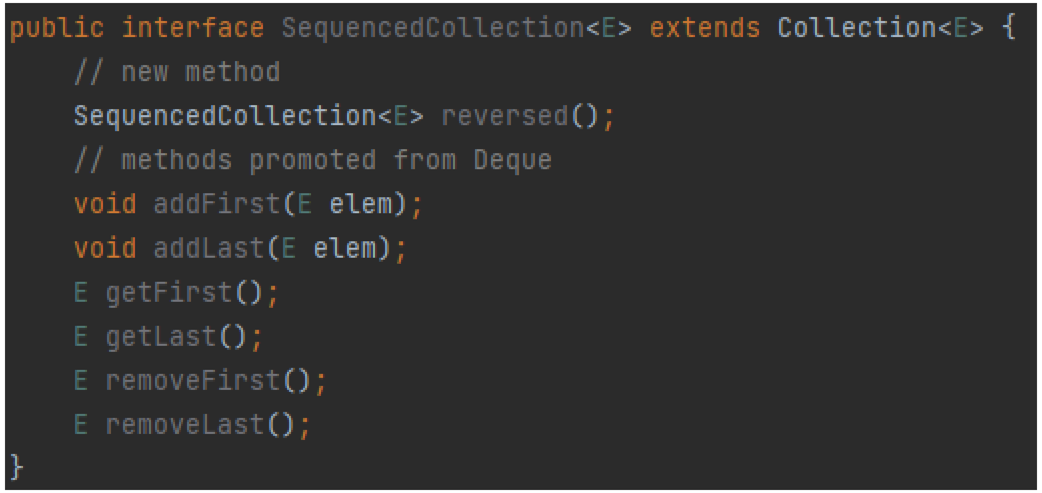
- Both subtasks share one returning point in the parent thread
- The policy (=implementation of StructuredTaskScope) dictates the behaviour;
- Try-with-resources -> cleanup of children when parent interrupts



```
static void handleOnSuccess() throws ExecutionException, InterruptedException {
    try (var scope = new StructuredTaskScope.ShutdownOnSuccess<>()) {
        scope.fork(() -> findUser());
        scope.fork(() -> fetchOrder());

        Object result = scope.join().result();

        if (result instanceof String s) {
            System.out.printf("Fetching user(%s) was a success!%n", s);
        } else if (result instanceof Integer i) {
            System.out.printf("Fetching order(%s) was a success!%n", i);
        }
    }
}
```

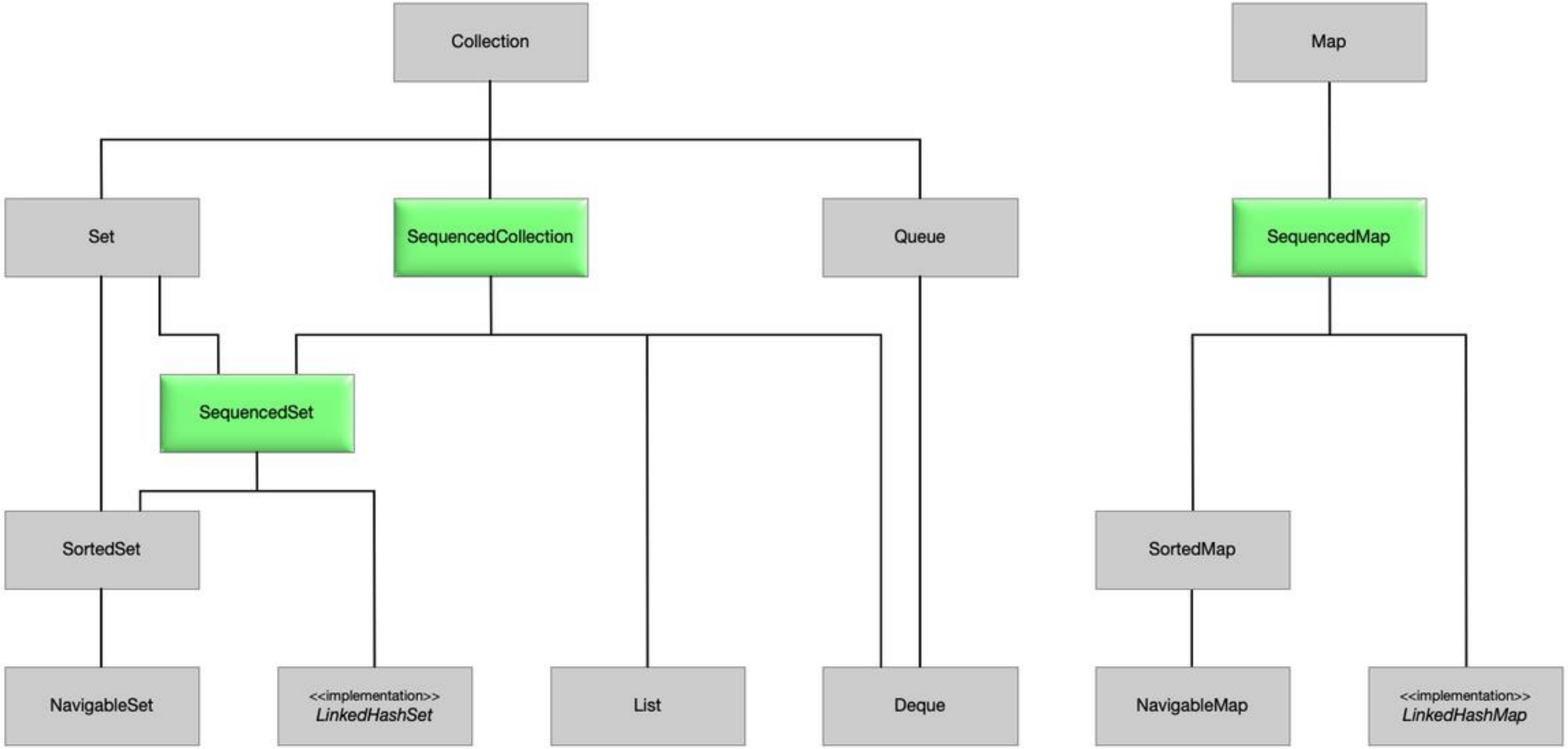
Java 21 – Sequenced Collections

- New interface for all Collection classes that have a notion of ‘order’
- Methods for **inserting, deleting and querying** at the head and tail of a collections



```java
public interface SequencedCollection<E> extends Collection<E> {
    // new method
    SequencedCollection<E> reversed();
    // methods promoted from Deque
    void addFirst(E elem);
    void addLast(E elem);
    E getFirst();
    E getLast();
    E removeFirst();
    E removeLast();
}
```

Java 21 – Sequenced Collections



Sequenced Collections JEP – Stuart Marks

2022-02-16