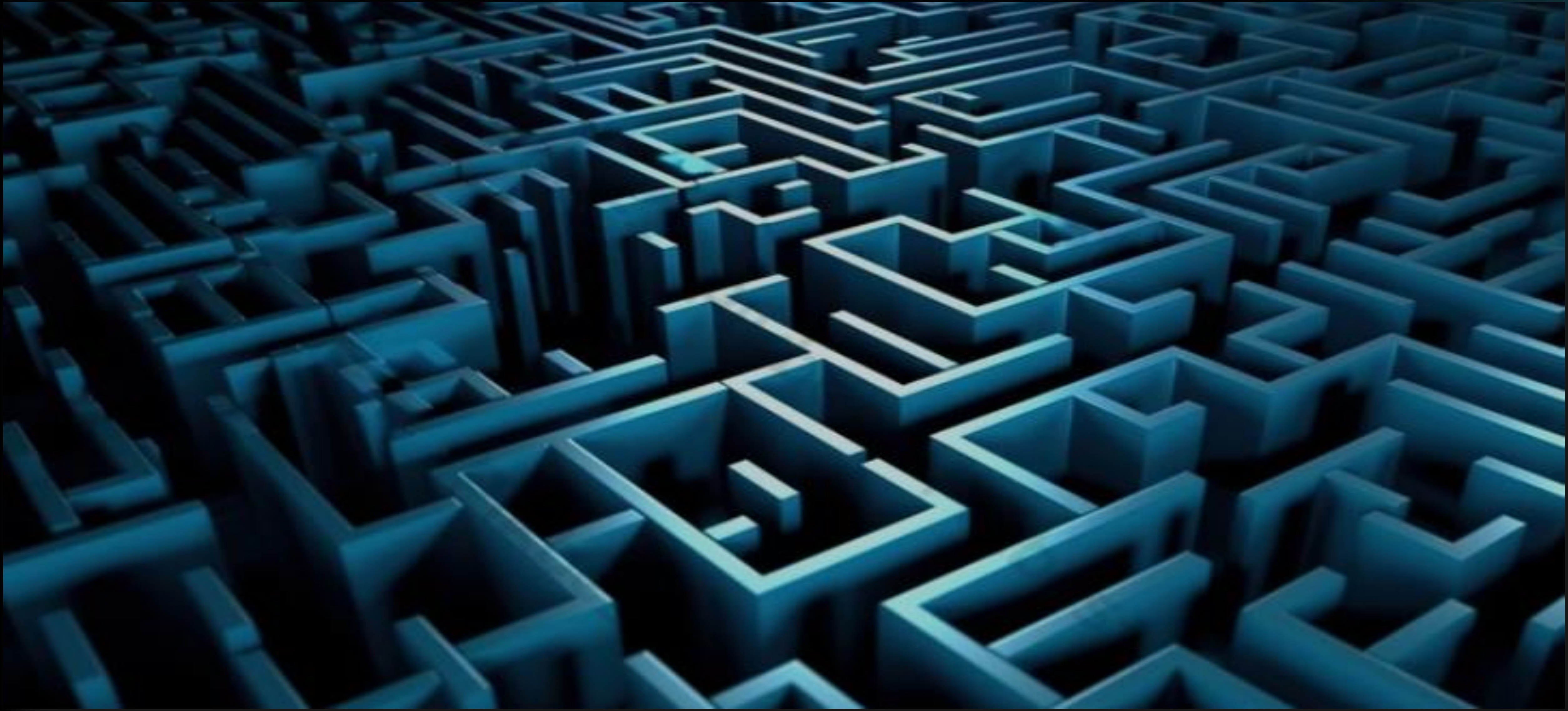
UNLOCKING JAVA'S CODE MAZE

Mihaela Gheorghe-Roman, PhD.
Software Architect

Unlocking JAVA’s code maze



Agenda

- Exercises!
  - Collections
  - Streams
  - Records
  - Concurrency
  - Optional
  - Many other Java gems!

Unlocking JAVA’s code maze

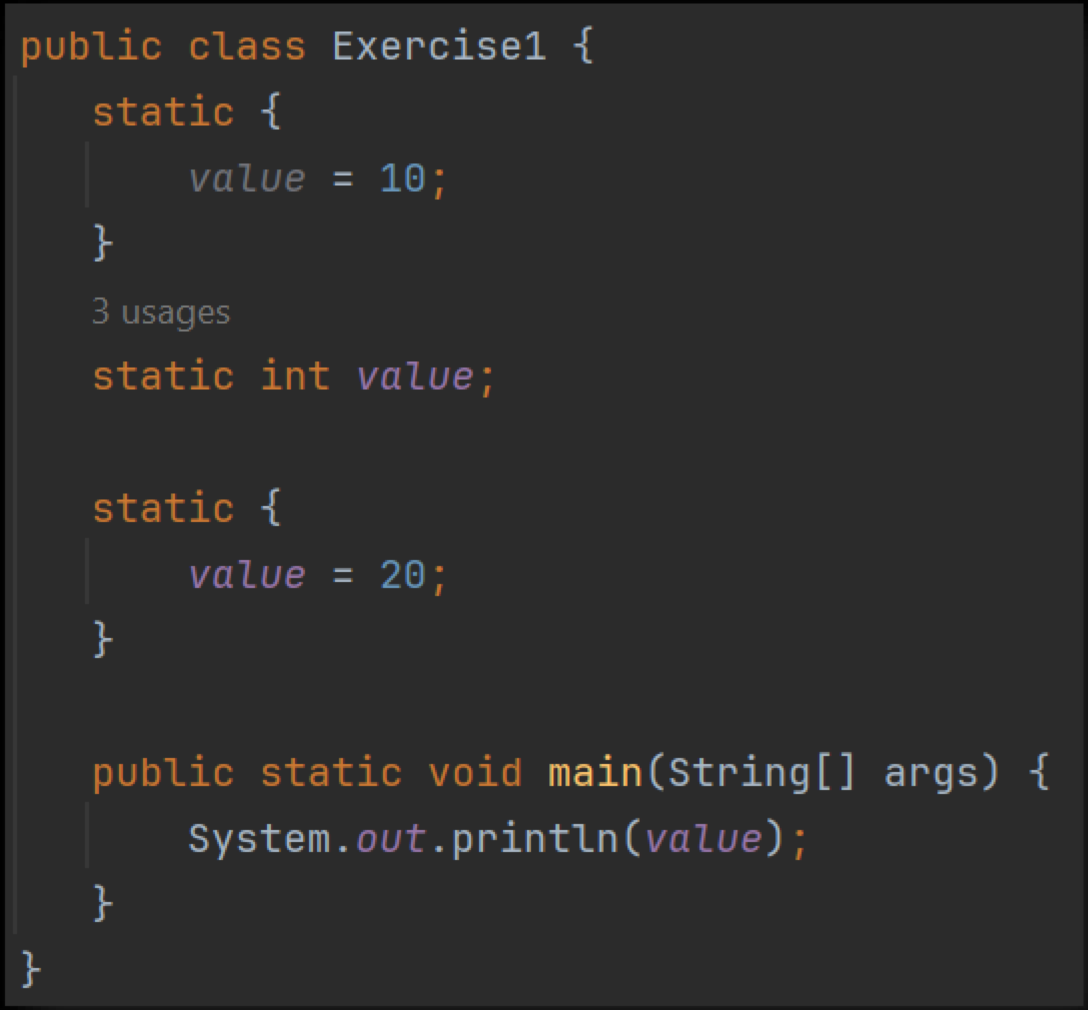
slido.com

#3509 467

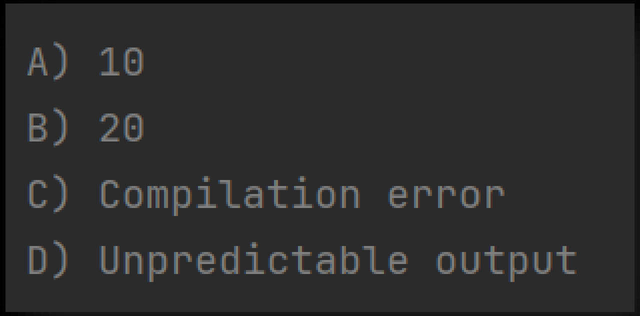
Unlocking JAVA’s code maze

Ready?

1: Static Initialization and Blocks

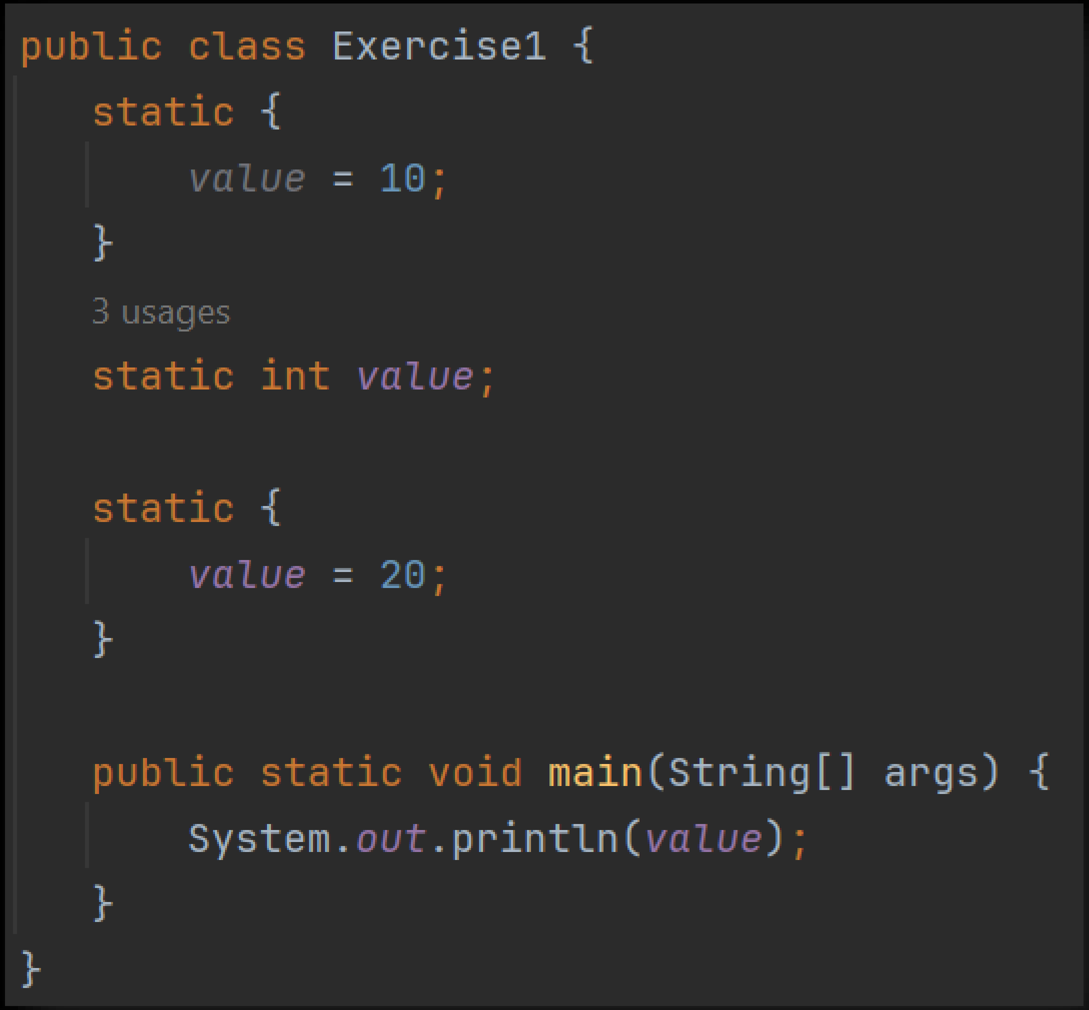


```java
public class Exercise1 {
    static {
        value = 10;
    }
    3 usages
    static int value;

    static {
        value = 20;
    }

    public static void main(String[] args) {
        System.out.println(value);
    }
}
```



```
A) 10
B) 20
C) Compilation error
D) Unpredictable output
```

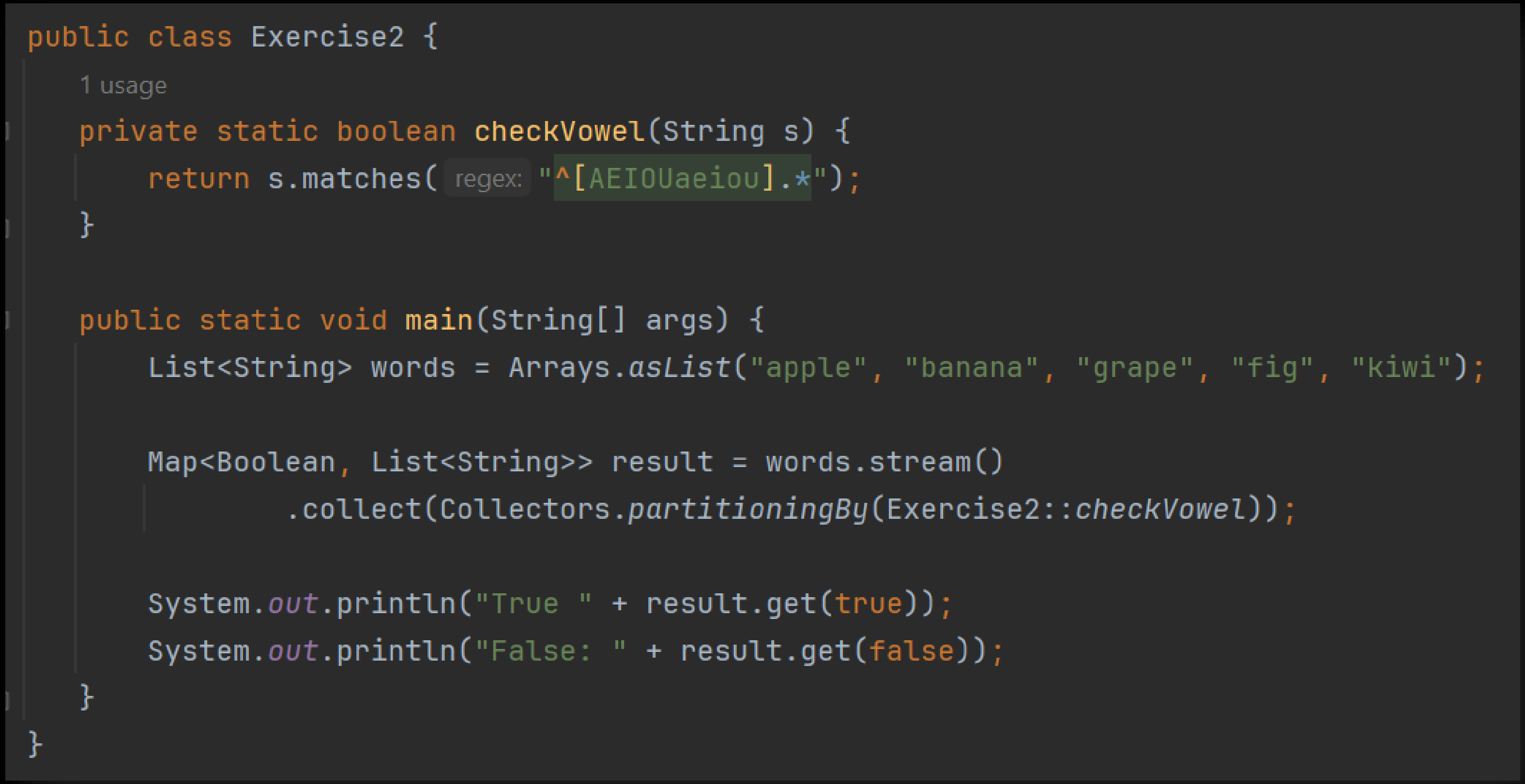
1: Static Initialization and Blocks



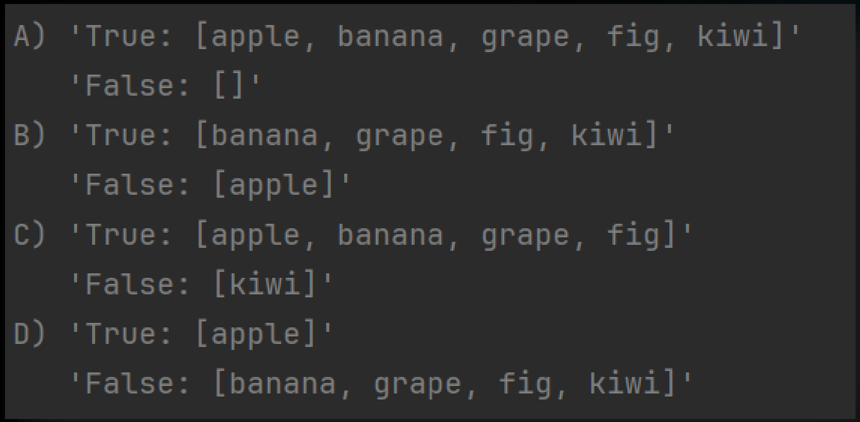
```
public class Exercise1 {
    static {
        value = 10;
    }
    3 usages
    static int value;

    static {
        value = 20;
    }

    public static void main(String[] args) {
        System.out.println(value);
    }
}
```

B) 20

2: Streams

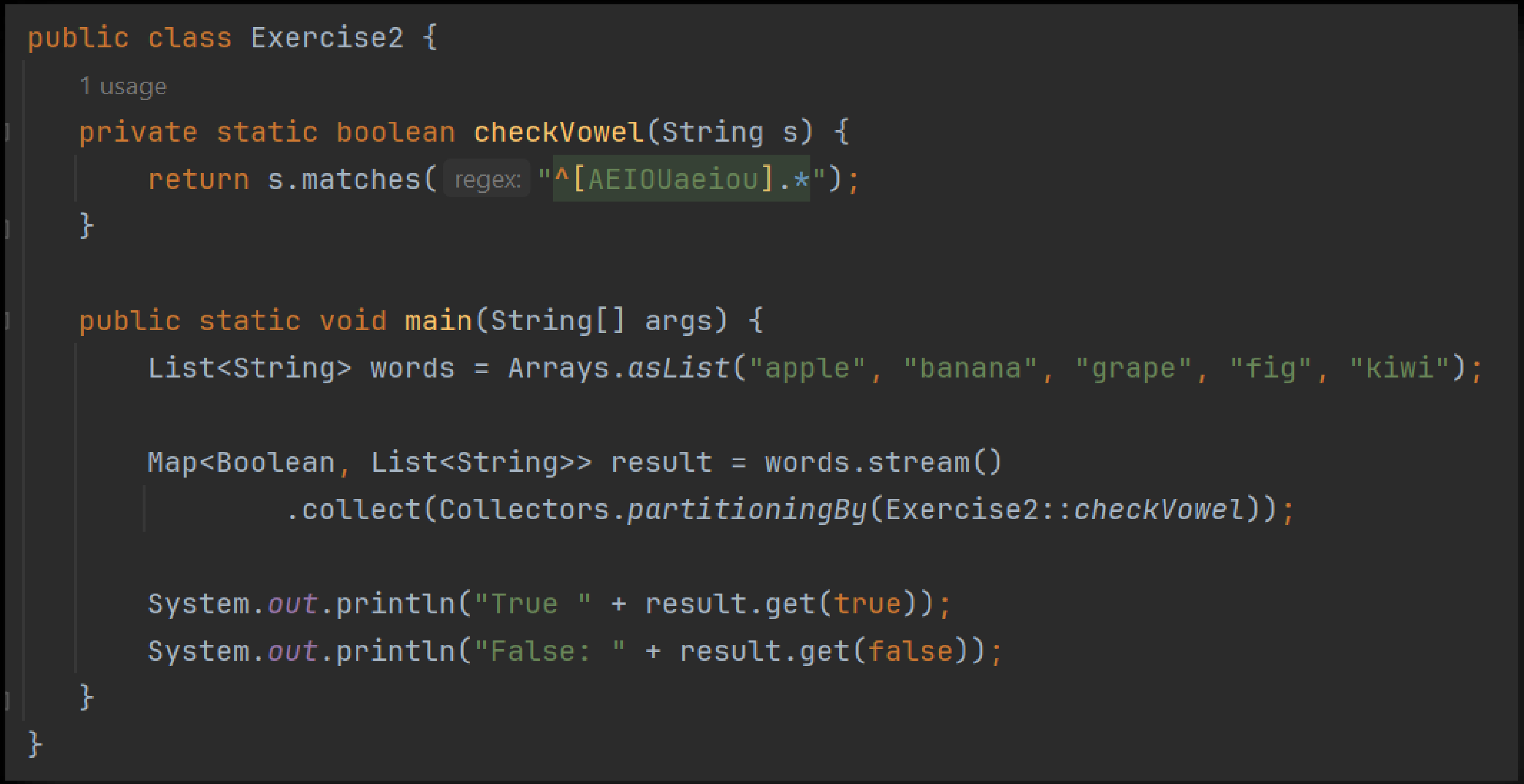


```
public class Exercise2 {
    1 usage
    private static boolean checkVowel(String s) {
        return s.matches( regex: "^[AEIOUaeiou].*");
    }

    public static void main(String[] args) {
        List<String> words = Arrays.asList("apple", "banana", "grape", "fig", "kiwi");

        Map<Boolean, List<String>> result = words.stream()
                .collect(Collectors.partitioningBy(Exercise2::checkVowel));

        System.out.println("True " + result.get(true));
        System.out.println("False: " + result.get(false));
    }
}
```

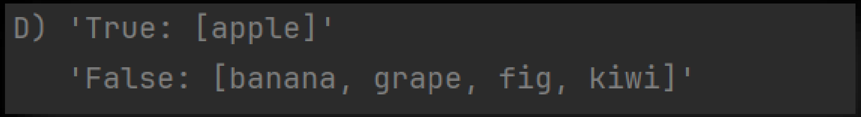


```
A) 'True: [apple, banana, grape, fig, kiwi]'
   'False: []'
B) 'True: [banana, grape, fig, kiwi]'
   'False: [apple]'
C) 'True: [apple, banana, grape, fig]'
   'False: [kiwi]'
D) 'True: [apple]'
   'False: [banana, grape, fig, kiwi]'
```

2: Streams

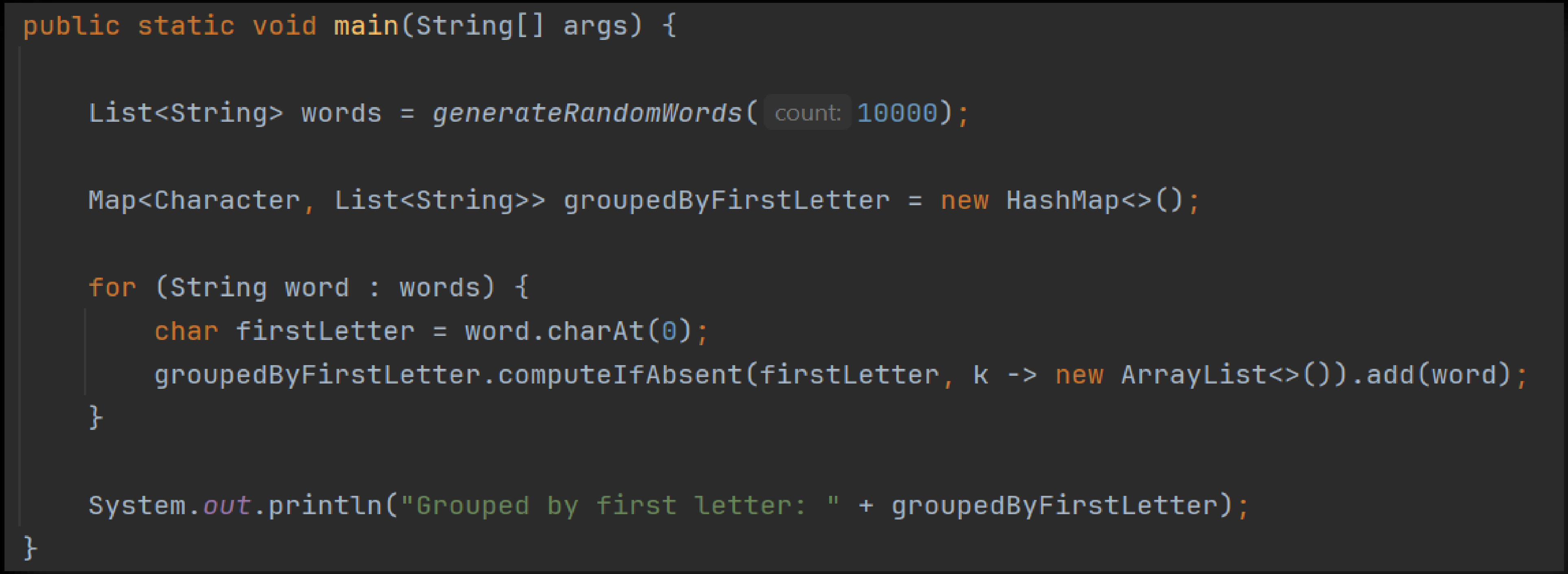


```java
public class Exercise2 {
    1 usage
    private static boolean checkVowel(String s) {
        return s.matches( regex: "^[AEIOUaeiou].*");
    }

    public static void main(String[] args) {
        List<String> words = Arrays.asList("apple", "banana", "grape", "fig", "kiwi");

        Map<Boolean, List<String>> result = words.stream()
                .collect(Collectors.partitioningBy(Exercise2::checkVowel));

        System.out.println("True " + result.get(true));
        System.out.println("False: " + result.get(false));
    }
}
```

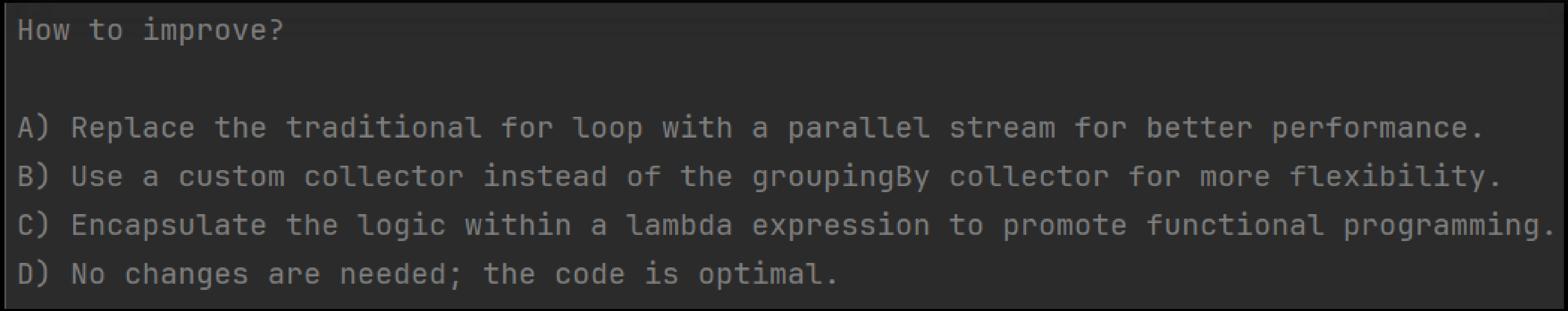


```
D) 'True: [apple]'
   'False: [banana, grape, fig, kiwi]'
```

3: Code optimization

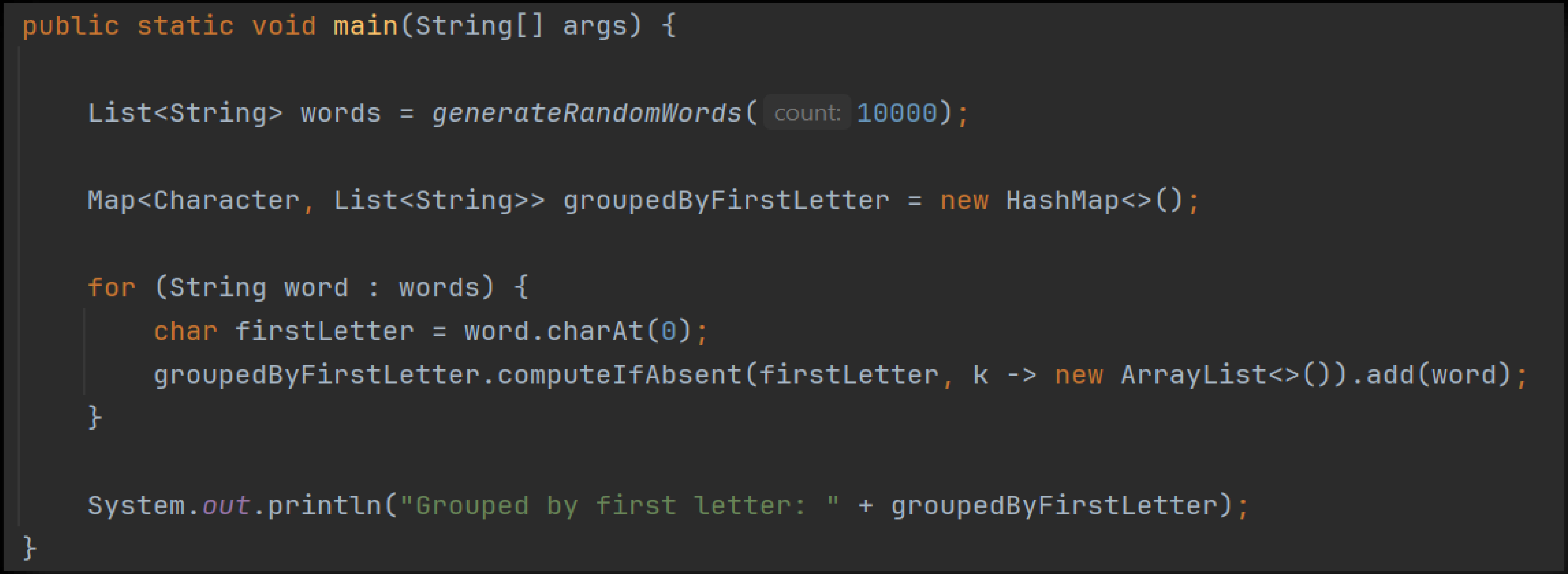


```java
public static void main(String[] args) {

    List<String> words = generateRandomWords( count: 10000);

    Map<Character, List<String>> groupedByFirstLetter = new HashMap<>();

    for (String word : words) {
        char firstLetter = word.charAt(0);
        groupedByFirstLetter.computeIfAbsent(firstLetter, k -> new ArrayList<>()).add(word);
    }

    System.out.println("Grouped by first letter: " + groupedByFirstLetter);
}
```

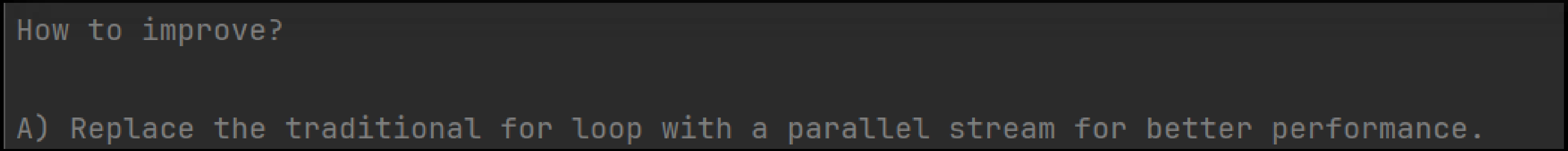


```
How to improve?

A) Replace the traditional for loop with a parallel stream for better performance.
B) Use a custom collector instead of the groupingBy collector for more flexibility.
C) Encapsulate the logic within a lambda expression to promote functional programming.
D) No changes are needed; the code is optimal.
```

3: Code optimization

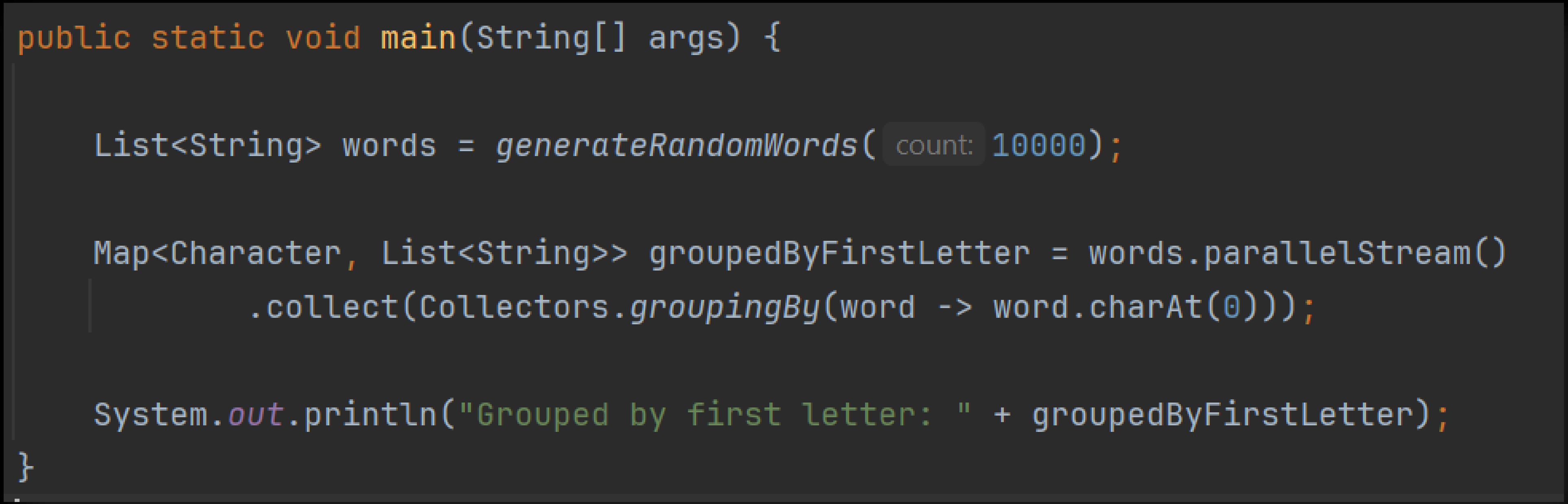


```
public static void main(String[] args) {

    List<String> words = generateRandomWords( count: 10000);

    Map<Character, List<String>> groupedByFirstLetter = new HashMap<>();

    for (String word : words) {
        char firstLetter = word.charAt(0);
        groupedByFirstLetter.computeIfAbsent(firstLetter, k -> new ArrayList<>()).add(word);
    }

    System.out.println("Grouped by first letter: " + groupedByFirstLetter);
}
```

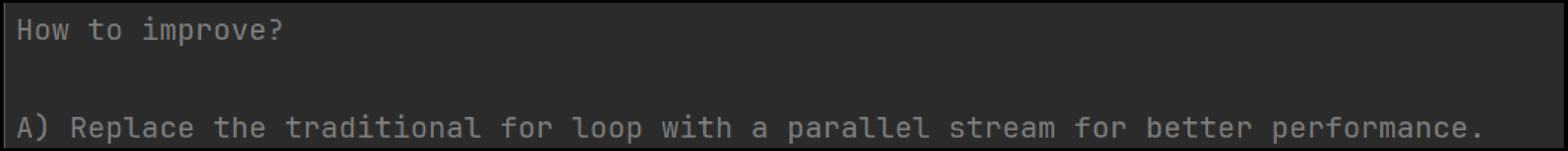


```
How to improve?


A) Replace the traditional for loop with a parallel stream for better performance.
```

3: Code optimization

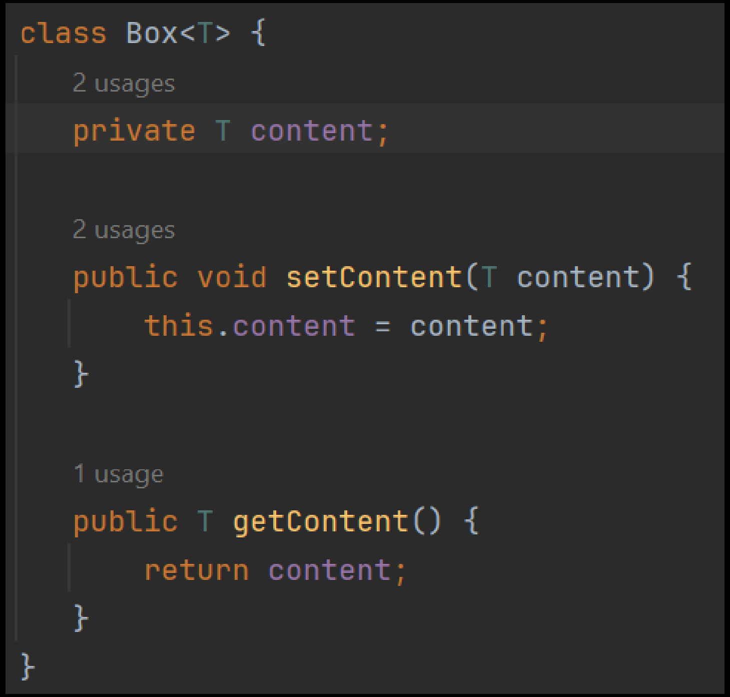


```java
public static void main(String[] args) {

    List<String> words = generateRandomWords( count: 10000);

    Map<Character, List<String>> groupedByFirstLetter = words.parallelStream()
            .collect(Collectors.groupingBy(word -> word.charAt(0)));

    System.out.println("Grouped by first letter: " + groupedByFirstLetter);
}
```

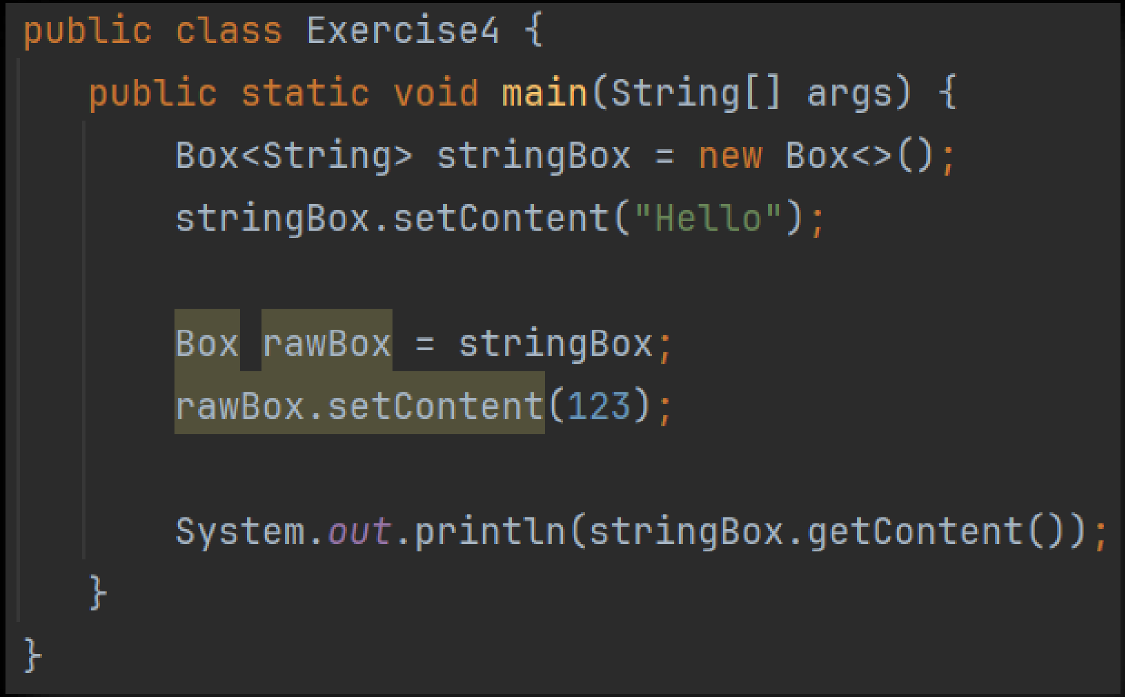


```
How to improve?

A) Replace the traditional for loop with a parallel stream for better performance.
```

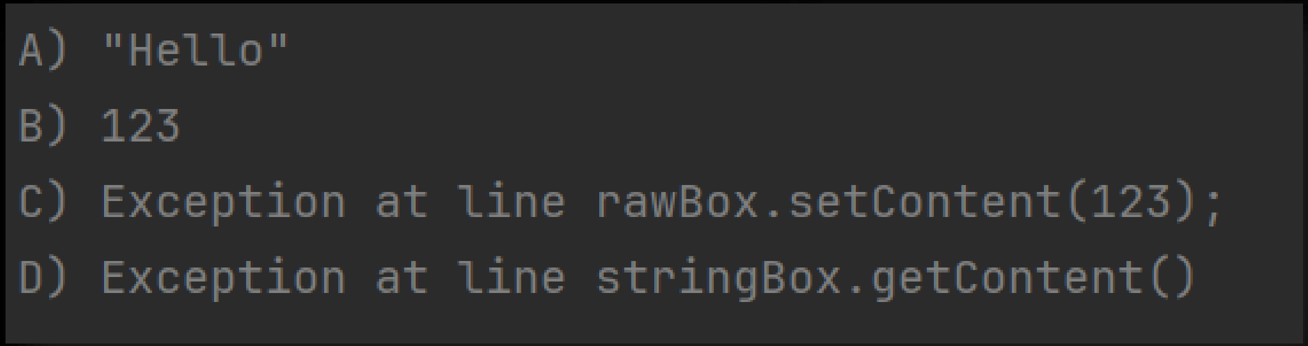
4: Type Erasure and Generic Constraints



```java
class Box<T> {
    2 usages
    private T content;

    2 usages
    public void setContent(T content) {
        this.content = content;
    }

    1 usage
    public T getContent() {
        return content;
    }
}
```

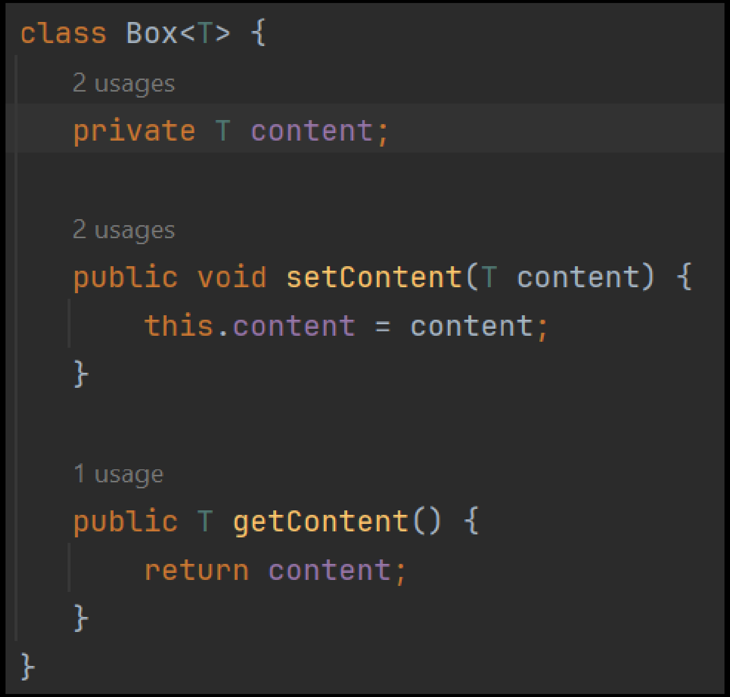


```java
public class Exercise4 {
    public static void main(String[] args) {
        Box<String> stringBox = new Box<>();
        stringBox.setContent("Hello");

        Box rawBox = stringBox;
        rawBox.setContent(123);

        System.out.println(stringBox.getContent());
    }
}
```

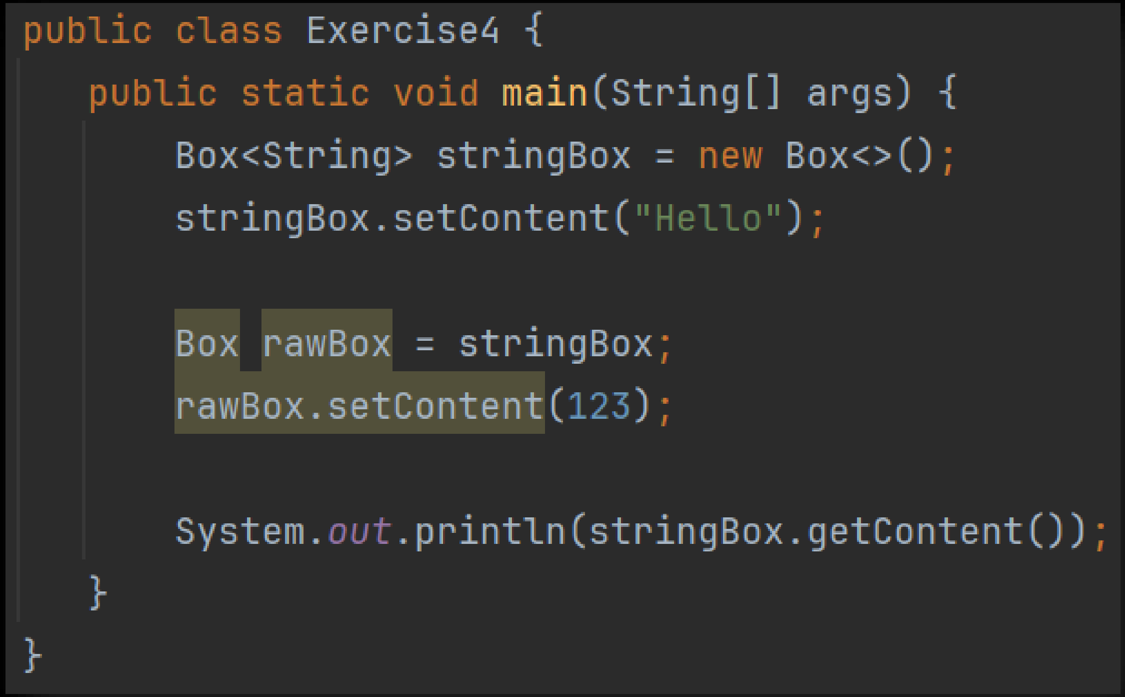


```
A) "Hello"
B) 123
C) Exception at line rawBox.setContent(123);
D) Exception at line stringBox.getContent()
```

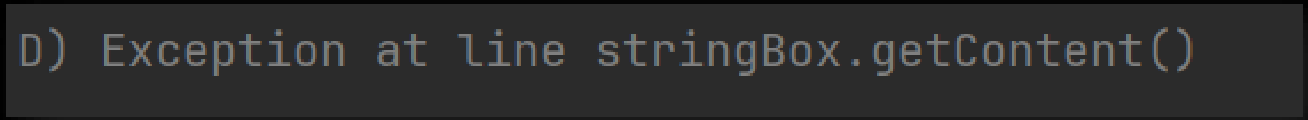
4: Type Erasure and Generic Constraints



```
class Box<T> {
    2 usages
    private T content;

    2 usages
    public void setContent(T content) {
        this.content = content;
    }

    1 usage
    public T getContent() {
        return content;
    }
}
```

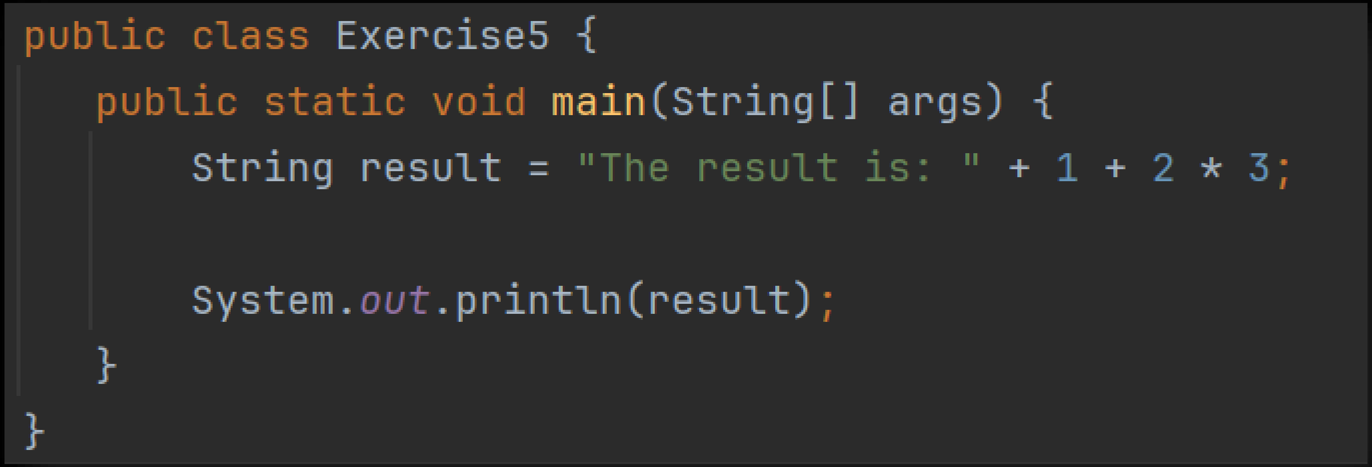


```
public class Exercise4 {
    public static void main(String[] args) {
        Box<String> stringBox = new Box<>();
        stringBox.setContent("Hello");

        Box rawBox = stringBox;
        rawBox.setContent(123);

        System.out.println(stringBox.getContent());
    }
}
```

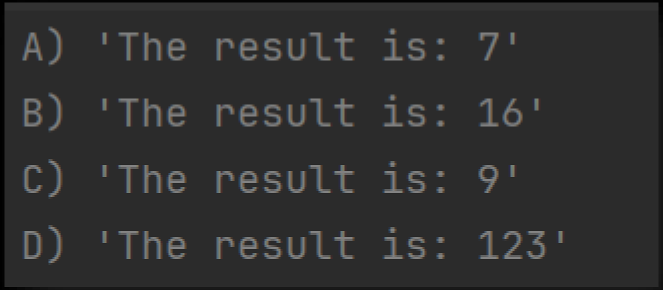


```
D) Exception at line stringBox.getContent()
```

5: String Concatenation

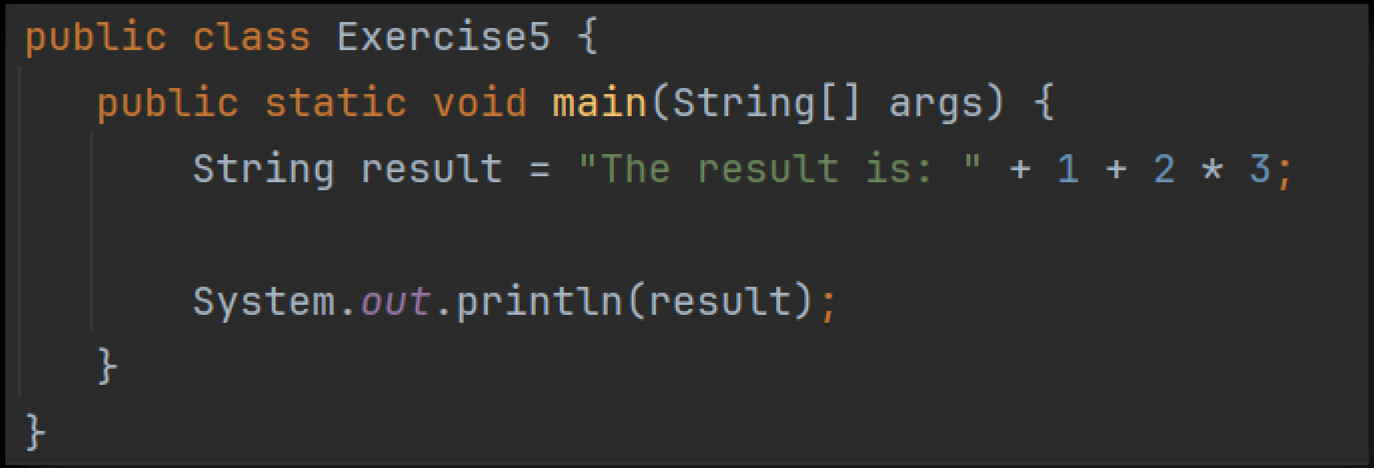


```java
public class Exercise5 {
   public static void main(String[] args) {
       String result = "The result is: " + 1 + 2 * 3;

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

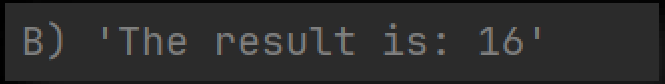


```
A) 'The result is: 7'
B) 'The result is: 16'
C) 'The result is: 9'
D) 'The result is: 123'
```

5: String Concatenation

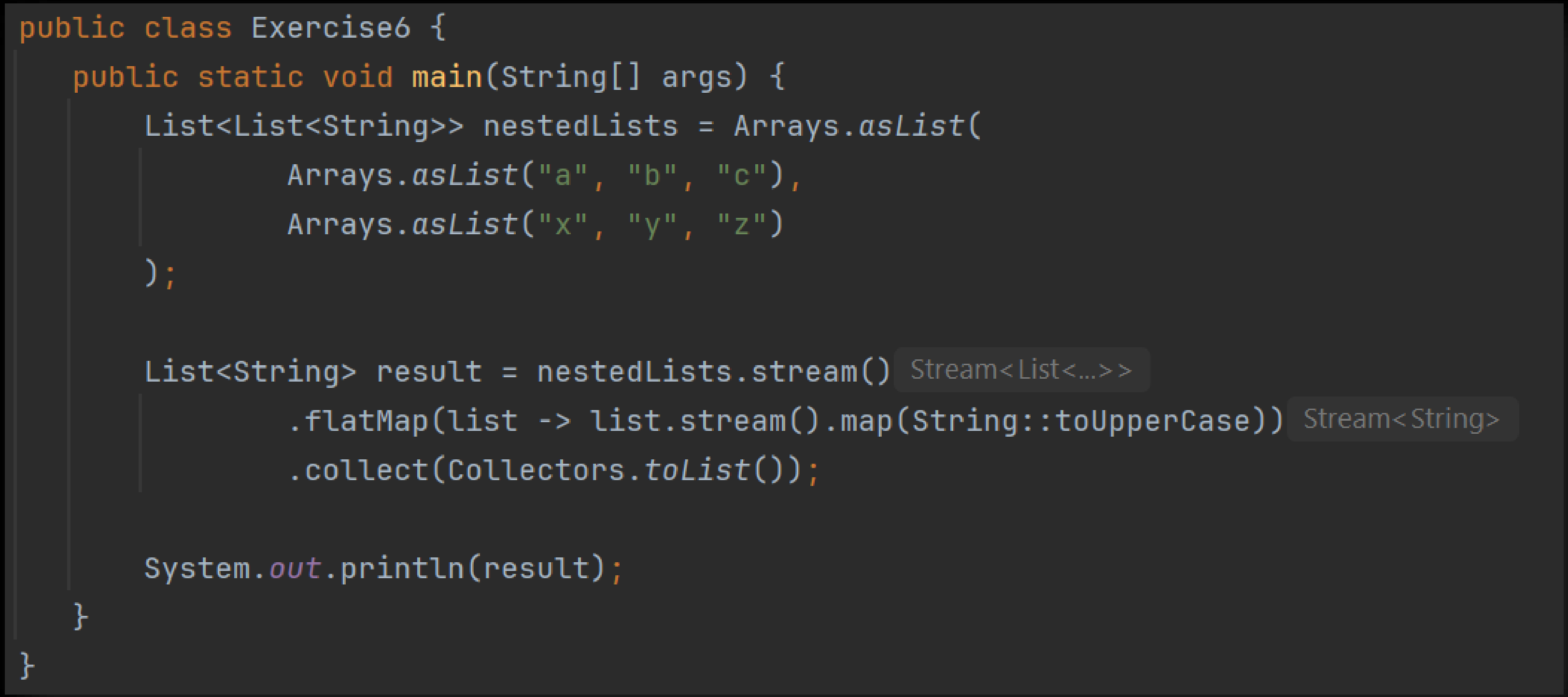


```java
public class Exercise5 {
    public static void main(String[] args) {
        String result = "The result is: " + 1 + 2 * 3;

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

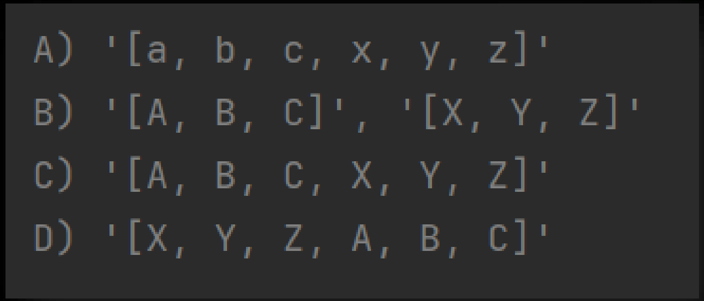


```
B) 'The result is: 16'
```

6: Mapping with Nested Streams

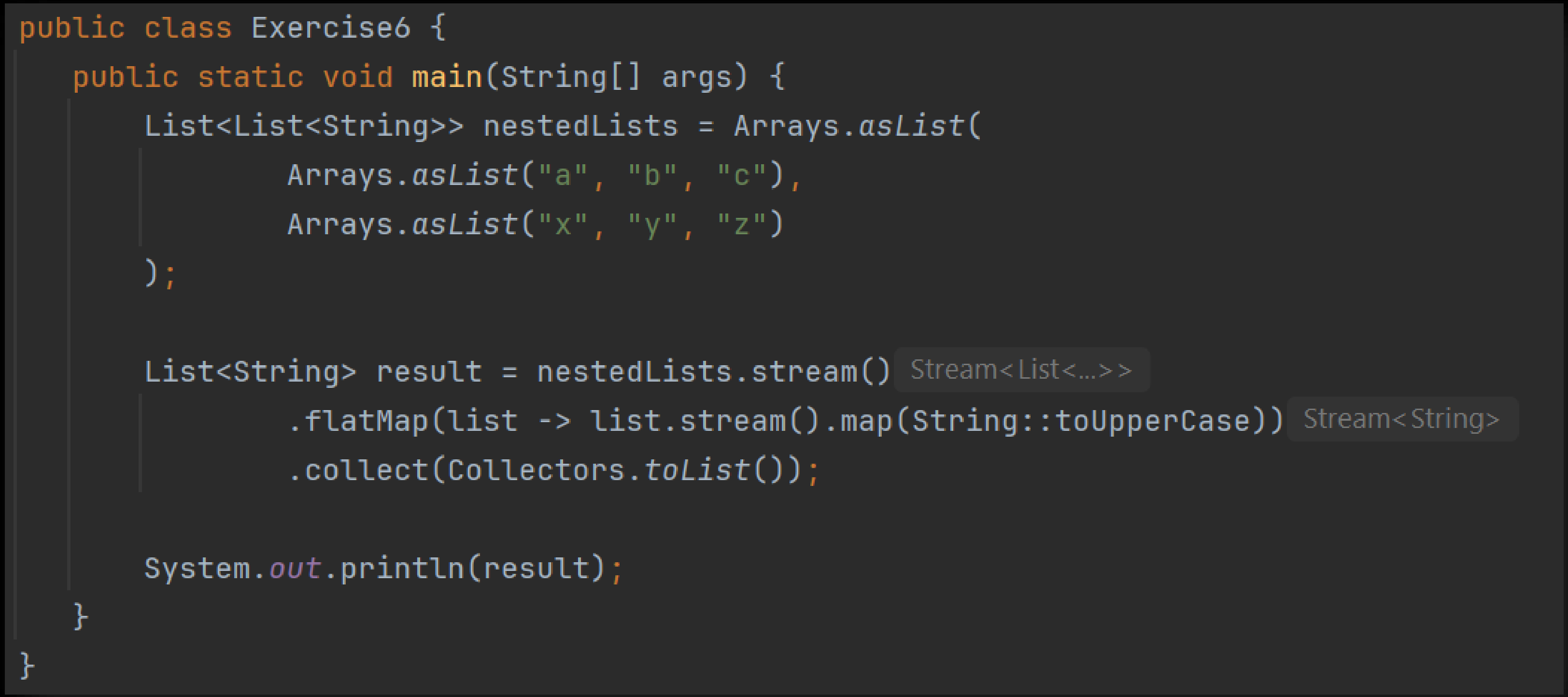


```
public class Exercise6 {
    public static void main(String[] args) {
        List<List<String>> nestedLists = Arrays.asList(
                Arrays.asList("a", "b", "c"),
                Arrays.asList("x", "y", "z")
        );

        List<String> result = nestedLists.stream() Stream<List<...>>
                .flatMap(list -> list.stream().map(String::toUpperCase)) Stream<String>
                .collect(Collectors.toList());

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

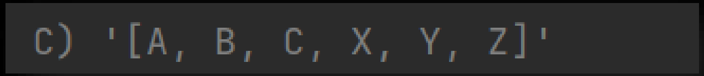


```
A) '[a, b, c, x, y, z]'
B) '[A, B, C]', '[X, Y, Z]'
C) '[A, B, C, X, Y, Z]'
D) '[X, Y, Z, A, B, C]'
```

6: Mapping with Nested Streams

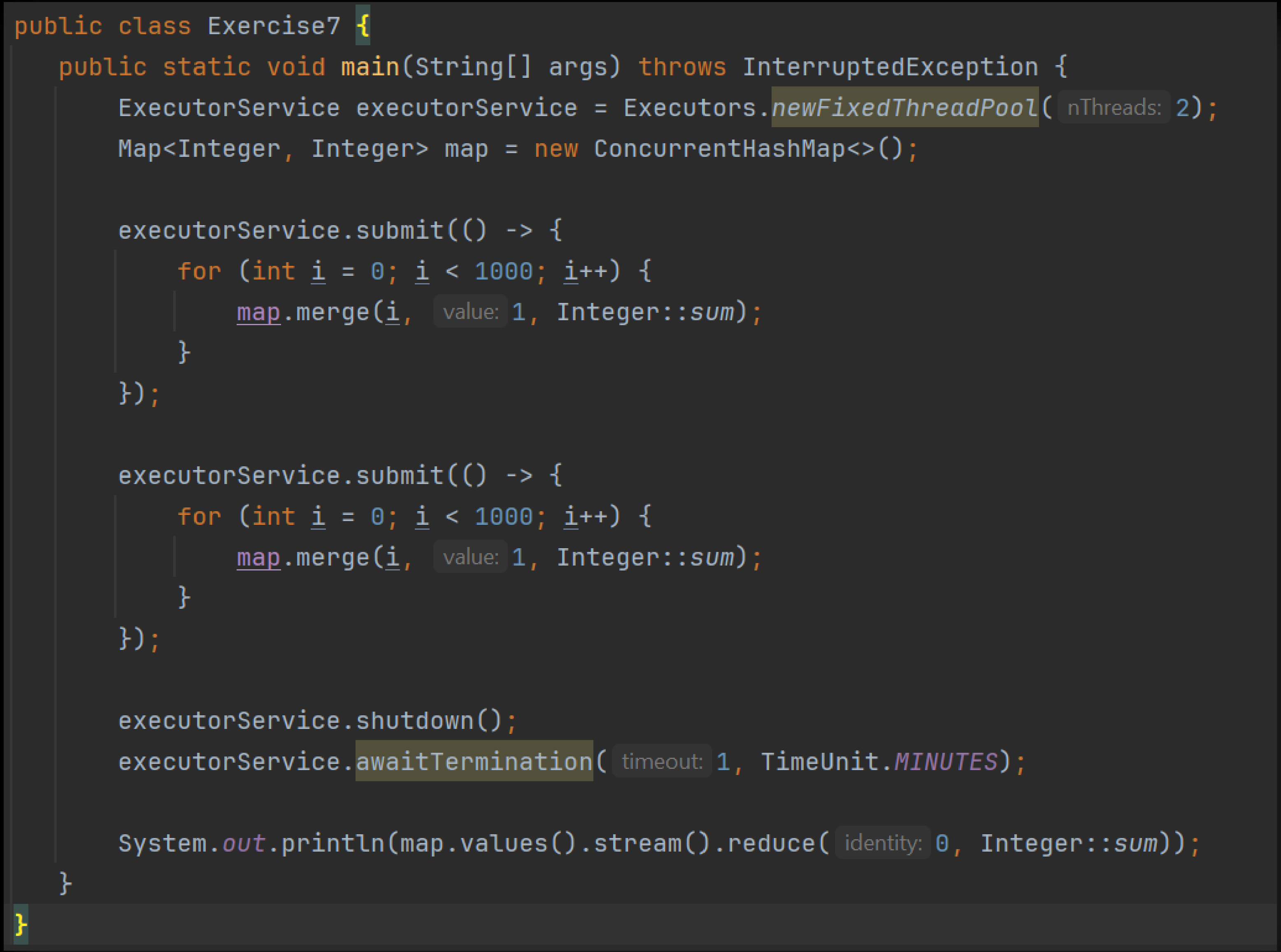


```java
public class Exercise6 {
    public static void main(String[] args) {
        List<List<String>> nestedLists = Arrays.asList(
                Arrays.asList("a", "b", "c"),
                Arrays.asList("x", "y", "z")
        );

        List<String> result = nestedLists.stream() Stream<List<...>>
                .flatMap(list -> list.stream().map(String::toUpperCase)) Stream<String>
                .collect(Collectors.toList());

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

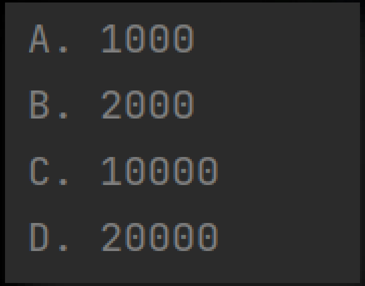


```
C) '[A, B, C, X, Y, Z]'
```

7: Java Concurrency

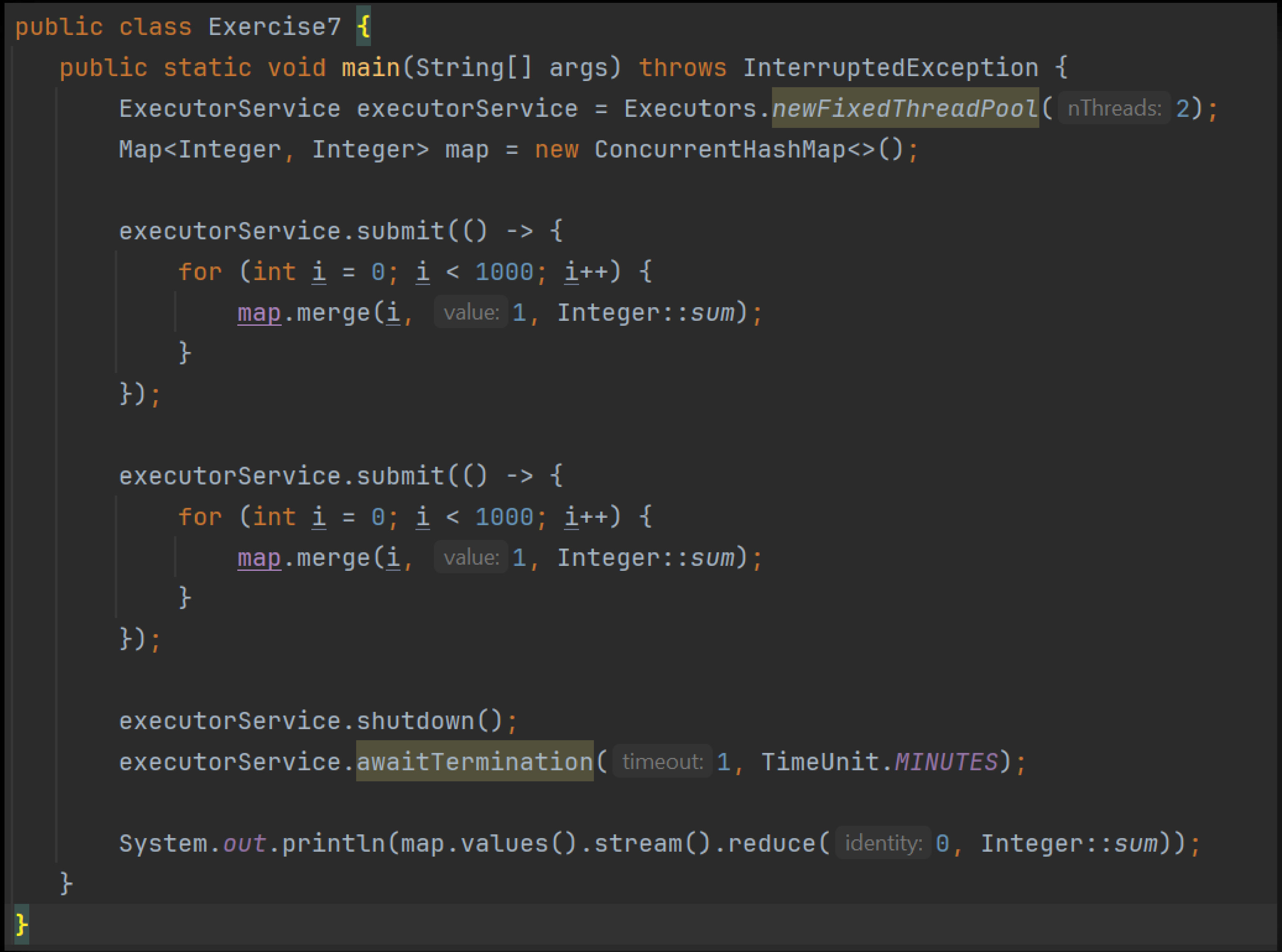


```
public class Exercise7 {
    public static void main(String[] args) throws InterruptedException {
        ExecutorService executorService = Executors.newFixedThreadPool( nThreads: 2);
        Map<Integer, Integer> map = new ConcurrentHashMap<>();

        executorService.submit(() -> {
            for (int i = 0; i < 1000; i++) {
                map.merge(i,  value: 1, Integer::sum);
            }
        });

        executorService.submit(() -> {
            for (int i = 0; i < 1000; i++) {
                map.merge(i,  value: 1, Integer::sum);
            }
        });

        executorService.shutdown();
        executorService.awaitTermination( timeout: 1, TimeUnit.MINUTES);

        System.out.println(map.values().stream().reduce( identity: 0, Integer::sum));
    }
}
```



```
A. 1000
B. 2000
C. 10000
D. 20000
```

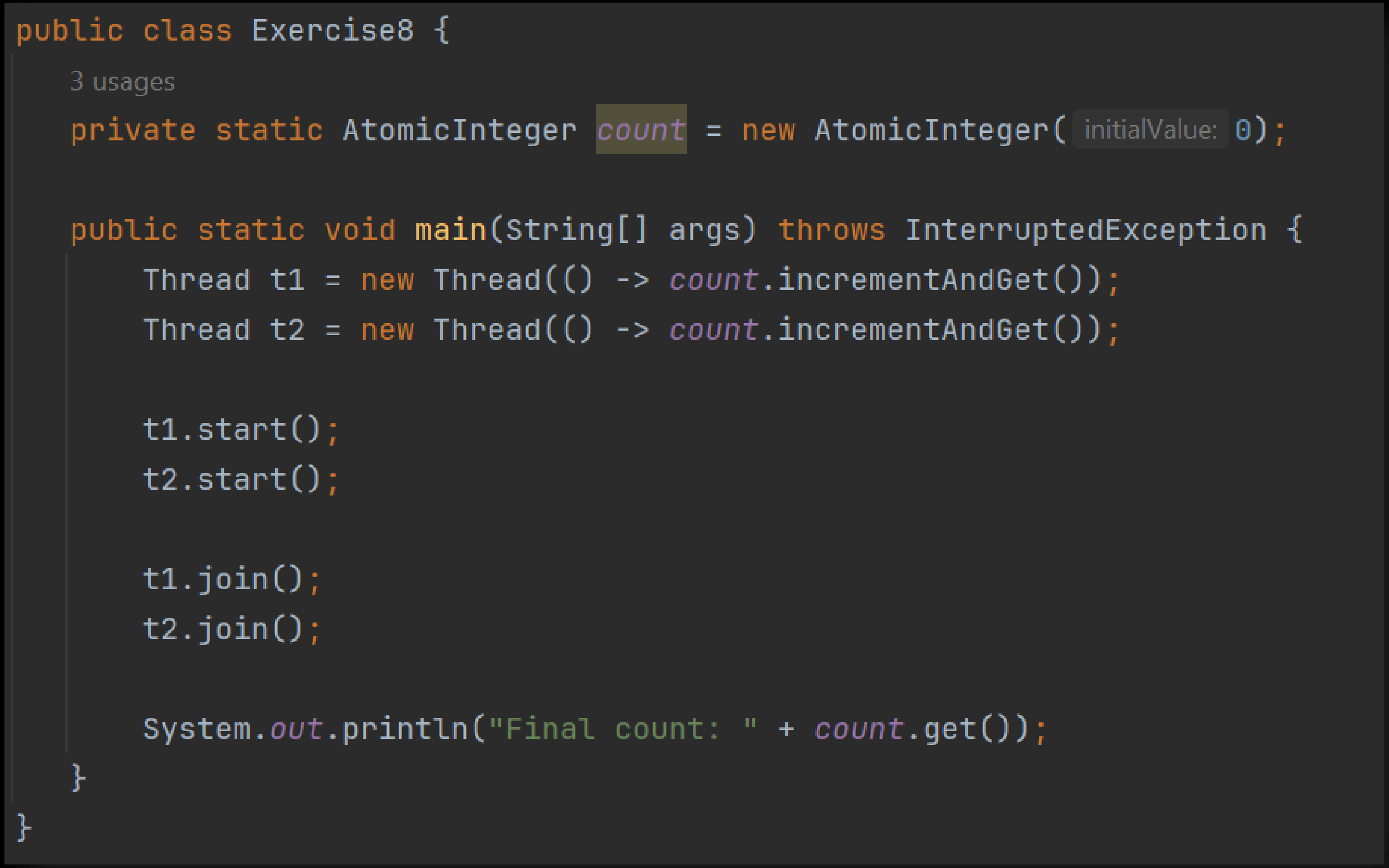
7: Java Concurrency



```
public class Exercise7 {
    public static void main(String[] args) throws InterruptedException {
        ExecutorService executorService = Executors.newFixedThreadPool( nThreads: 2);
        Map<Integer, Integer> map = new ConcurrentHashMap<>();

        executorService.submit(() -> {
            for (int i = 0; i < 1000; i++) {
                map.merge(i,  value: 1, Integer::sum);
            }
        });

        executorService.submit(() -> {
            for (int i = 0; i < 1000; i++) {
                map.merge(i,  value: 1, Integer::sum);
            }
        });

        executorService.shutdown();
        executorService.awaitTermination( timeout: 1, TimeUnit.MINUTES);

        System.out.println(map.values().stream().reduce( identity: 0, Integer::sum));
    }
}
```

B. 2000

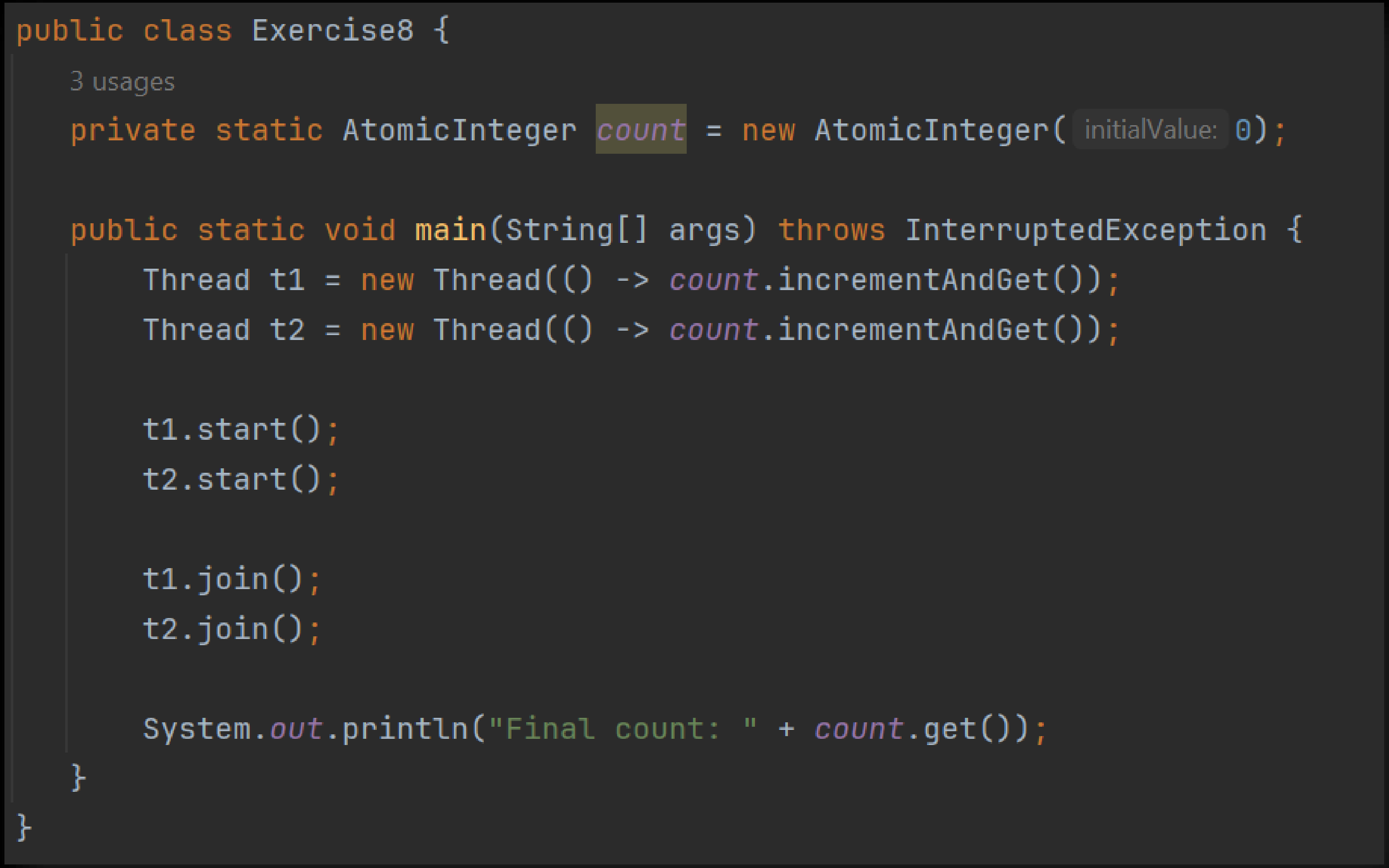
8: Concurrent Access with Atomic Variables



```
public class Exercise8 {
    3 usages
    private static AtomicInteger count = new AtomicInteger( initialValue: 0);

    public static void main(String[] args) throws InterruptedException {
        Thread t1 = new Thread(() -> count.incrementAndGet());
        Thread t2 = new Thread(() -> count.incrementAndGet());

        t1.start();
        t2.start();

        t1.join();
        t2.join();

        System.out.println("Final count: " + count.get());
    }
}
```

```
A) 'Final count: 2'
B) 'Final count: 1'
C) 'Final count: 0'
D) 'Error'
```

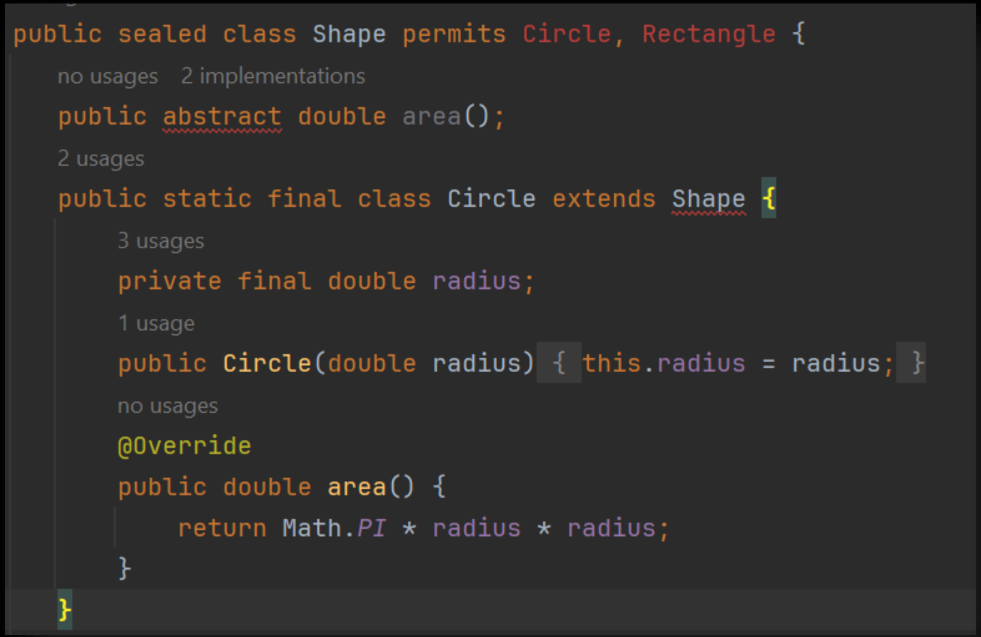
8: Concurrent Access with Atomic Variables



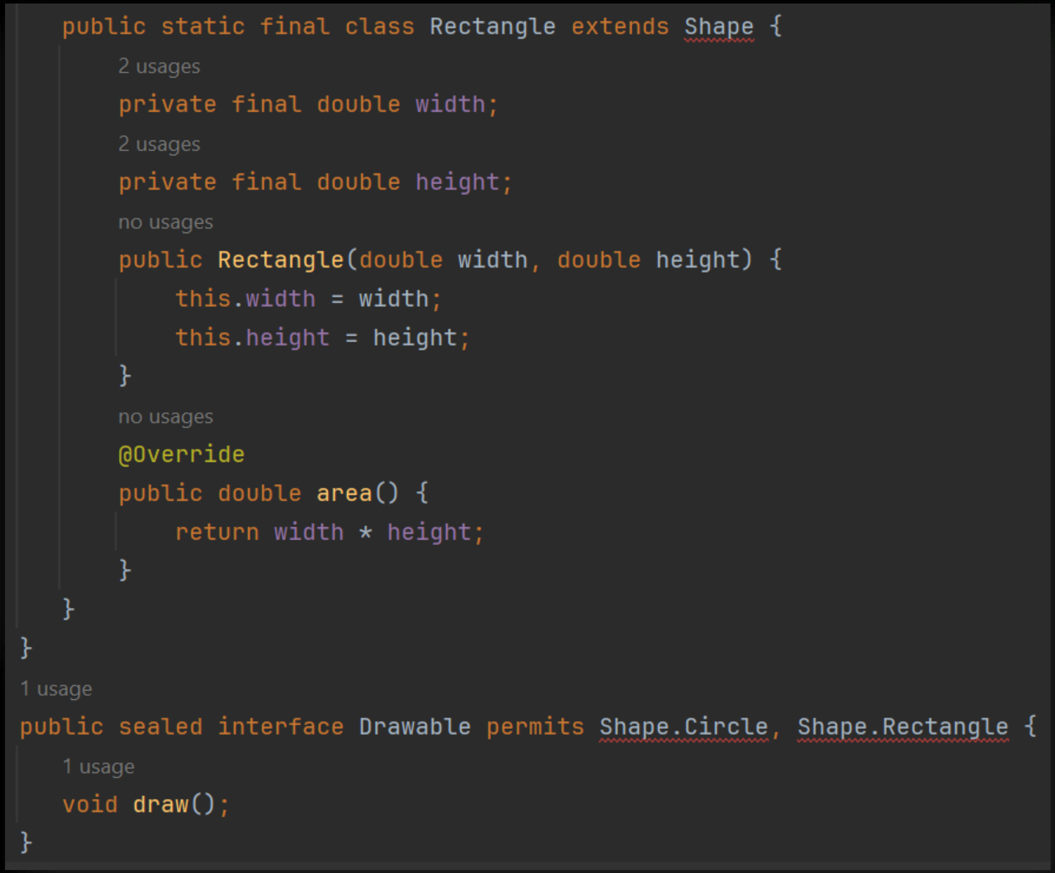
```
public class Exercise8 {
    3 usages
    private static AtomicInteger count = new AtomicInteger( initialValue: 0);

    public static void main(String[] args) throws InterruptedException {
        Thread t1 = new Thread(() -> count.incrementAndGet());
        Thread t2 = new Thread(() -> count.incrementAndGet());

        t1.start();
        t2.start();

        t1.join();
        t2.join();

        System.out.println("Final count: " + count.get());
    }
}
```

```
A) 'Final count: 2'
```

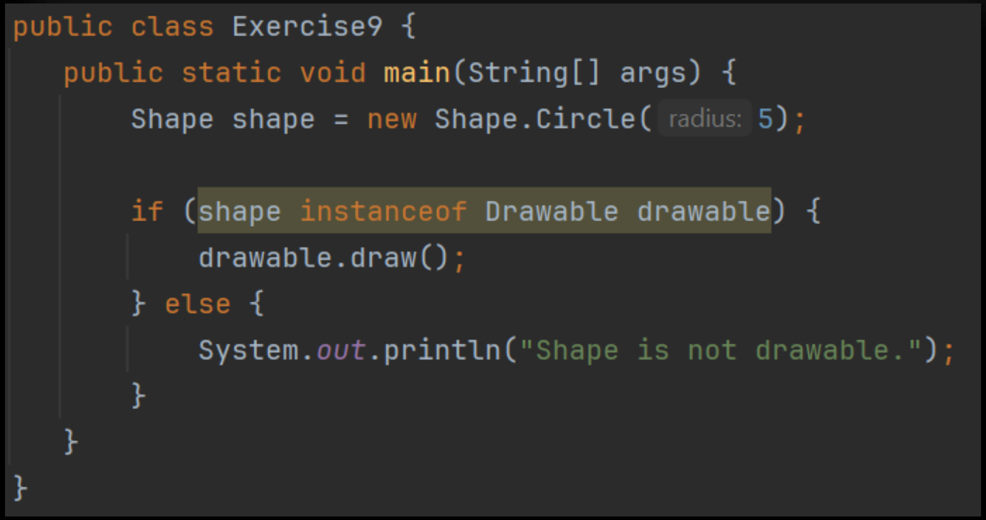
9: Sealed Classes



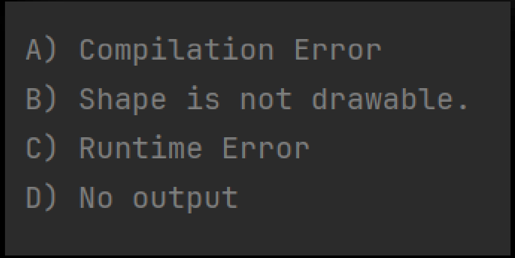
```
public sealed class Shape permits Circle, Rectangle {
    no usages   2 implementations
    public abstract double area();
    2 usages
    public static final class Circle extends Shape {
        3 usages
        private final double radius;
        1 usage
        public Circle(double radius) { this.radius = radius; }
        no usages
        @Override
        public double area() {
            return Math.PI * radius * radius;
        }
    }
```



```
    public static final class Rectangle extends Shape {
        2 usages
        private final double width;
        2 usages
        private final double height;
        no usages
        public Rectangle(double width, double height) {
            this.width = width;
            this.height = height;
        }
        no usages
        @Override
        public double area() {
            return width * height;
        }
    }
}
1 usage
public sealed interface Drawable permits Shape.Circle, Shape.Rectangle {
    1 usage
    void draw();
}
```

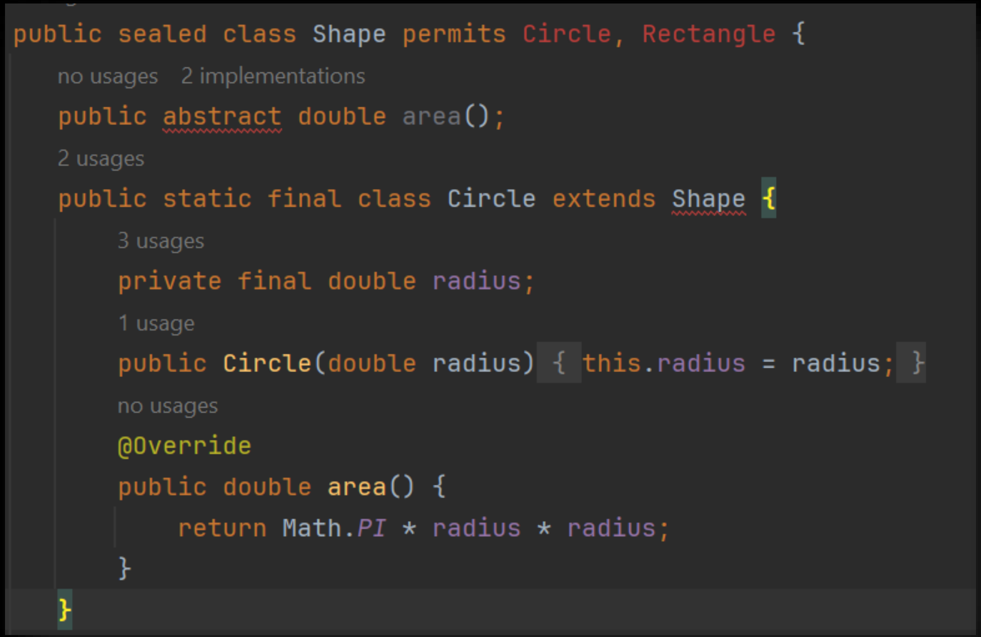


```
public class Exercise9 {
    public static void main(String[] args) {
        Shape shape = new Shape.Circle( radius: 5);

        if (shape instanceof Drawable drawable) {
            drawable.draw();
        } else {
            System.out.println("Shape is not drawable.");
        }
    }
}
```

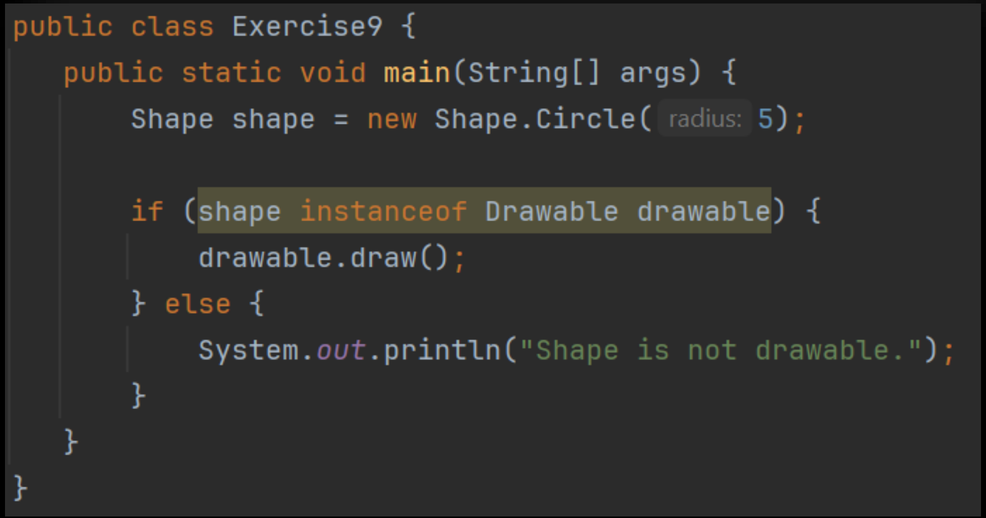


```
A) Compilation Error
B) Shape is not drawable.
C) Runtime Error
D) No output
```

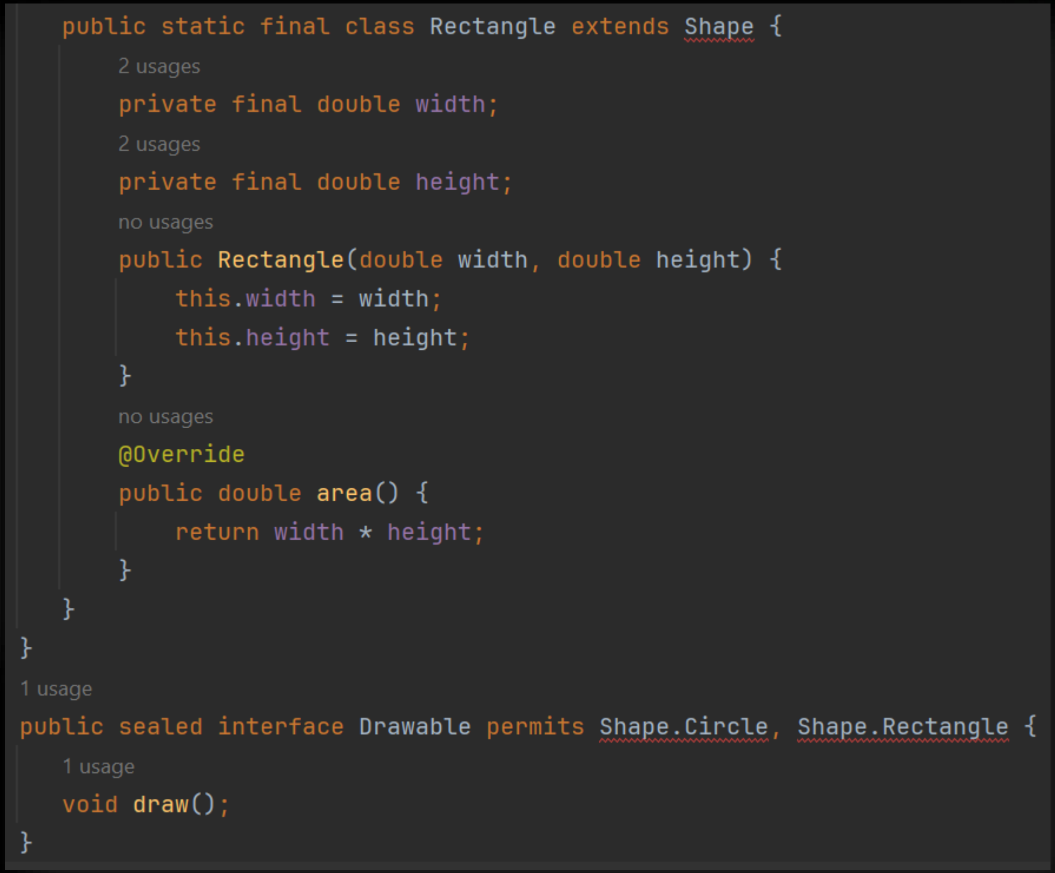
9: Sealed Classes



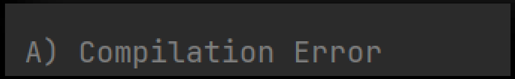
```
public sealed class Shape permits Circle, Rectangle {
    no usages   2 implementations
    public abstract double area();
    2 usages
    public static final class Circle extends Shape {
        3 usages
        private final double radius;
        1 usage
        public Circle(double radius) { this.radius = radius; }
        no usages
        @Override
        public double area() {
            return Math.PI * radius * radius;
        }
    }
```



```
public class Exercise9 {
    public static void main(String[] args) {
        Shape shape = new Shape.Circle( radius: 5);

        if (shape instanceof Drawable drawable) {
            drawable.draw();
        } else {
            System.out.println("Shape is not drawable.");
        }
    }
}
```

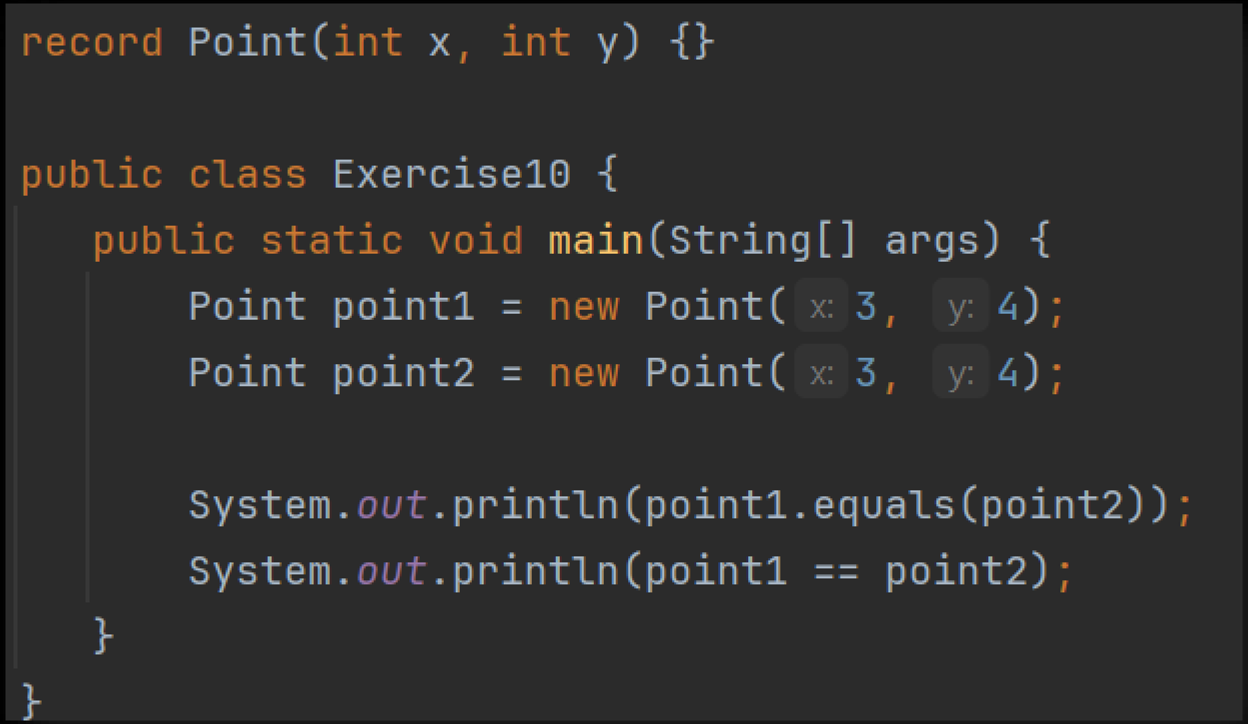


```
    public static final class Rectangle extends Shape {
        2 usages
        private final double width;
        2 usages
        private final double height;
        no usages
        public Rectangle(double width, double height) {
            this.width = width;
            this.height = height;
        }
        no usages
        @Override
        public double area() {
            return width * height;
        }
    }
}
1 usage
public sealed interface Drawable permits Shape.Circle, Shape.Rectangle {
    1 usage
    void draw();
}
```

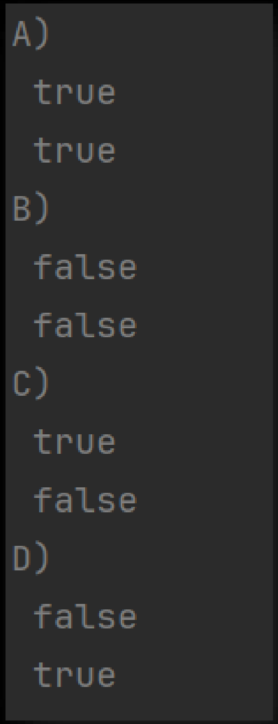


```
A) Compilation Error
```

10: Records

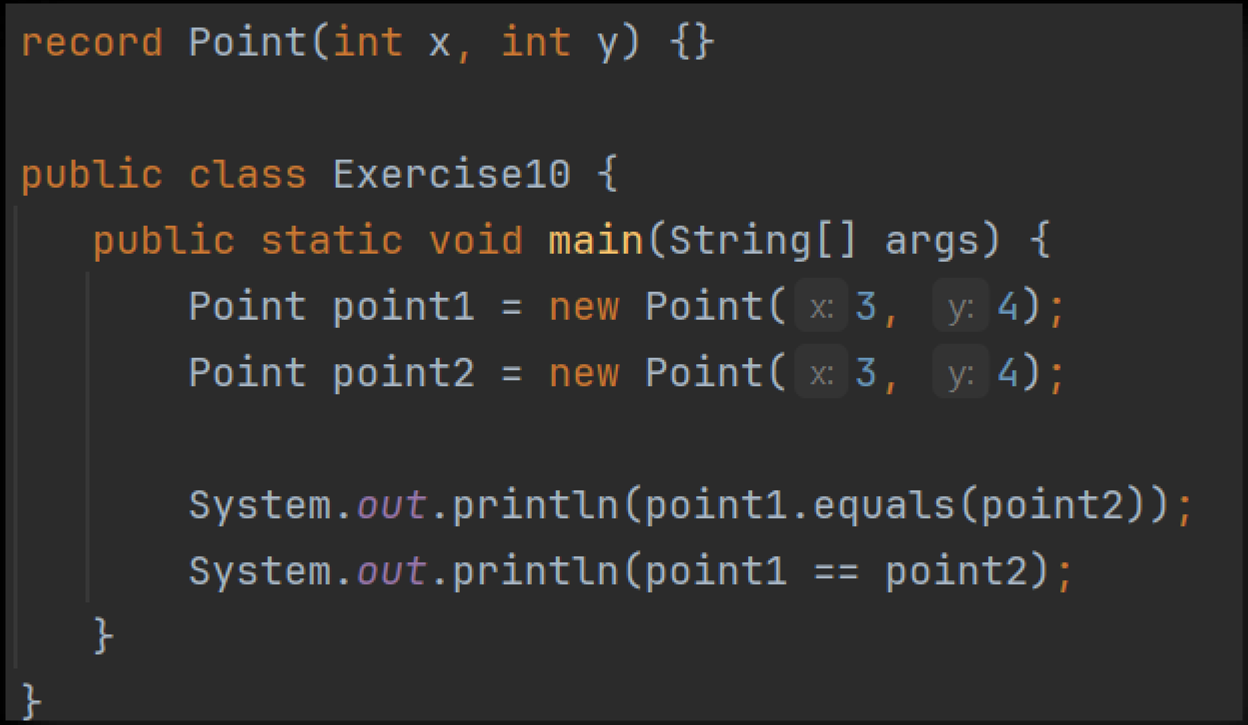


```
record Point(int x, int y) {}

public class Exercise10 {
    public static void main(String[] args) {
        Point point1 = new Point( x: 3,  y: 4);
        Point point2 = new Point( x: 3,  y: 4);

        System.out.println(point1.equals(point2));
        System.out.println(point1 == point2);
    }
}
```

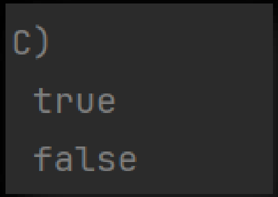


```
A)
  true
  true
B)
  false
  false
C)
  true
  false
D)
  false
  true
```

10: Records

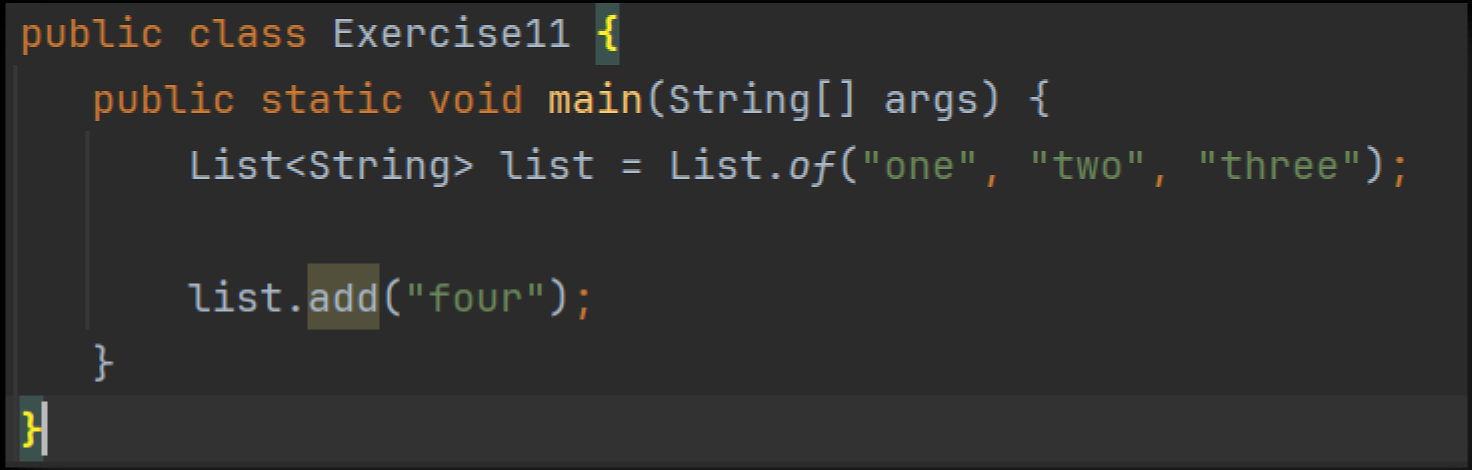


```
record Point(int x, int y) {}

public class Exercise10 {
    public static void main(String[] args) {
        Point point1 = new Point( x: 3,  y: 4);
        Point point2 = new Point( x: 3,  y: 4);

        System.out.println(point1.equals(point2));
        System.out.println(point1 == point2);
    }
}
```

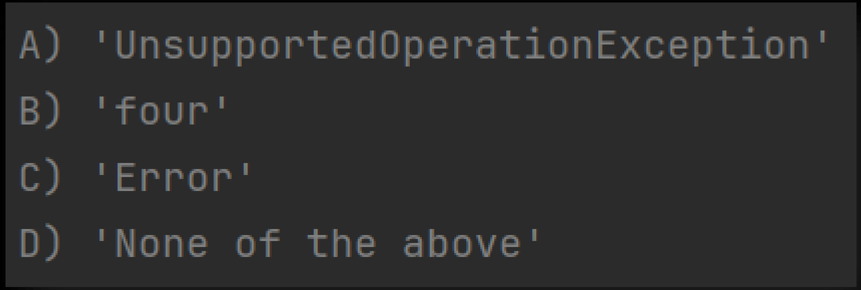


```
C)
 true
 false
```

11: List.of

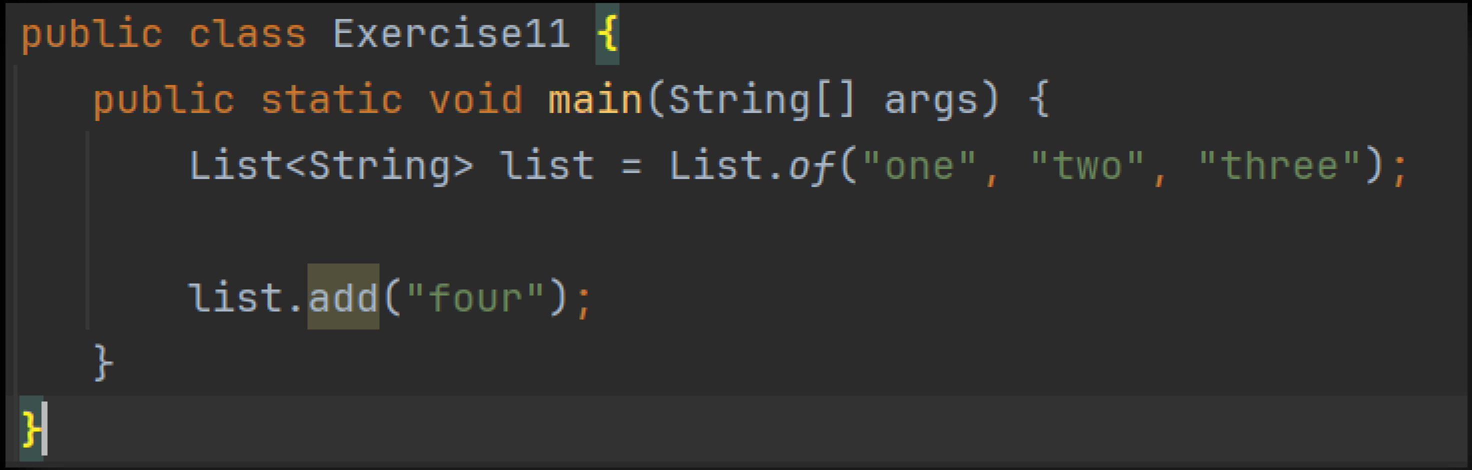


```
public class Exercise11 {
    public static void main(String[] args) {
        List<String> list = List.of("one", "two", "three");

        list.add("four");
    }
}
```

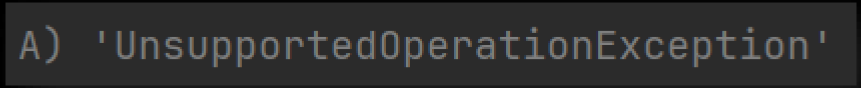


```
A) 'UnsupportedOperationException'
B) 'four'
C) 'Error'
D) 'None of the above'
```

11: List.of

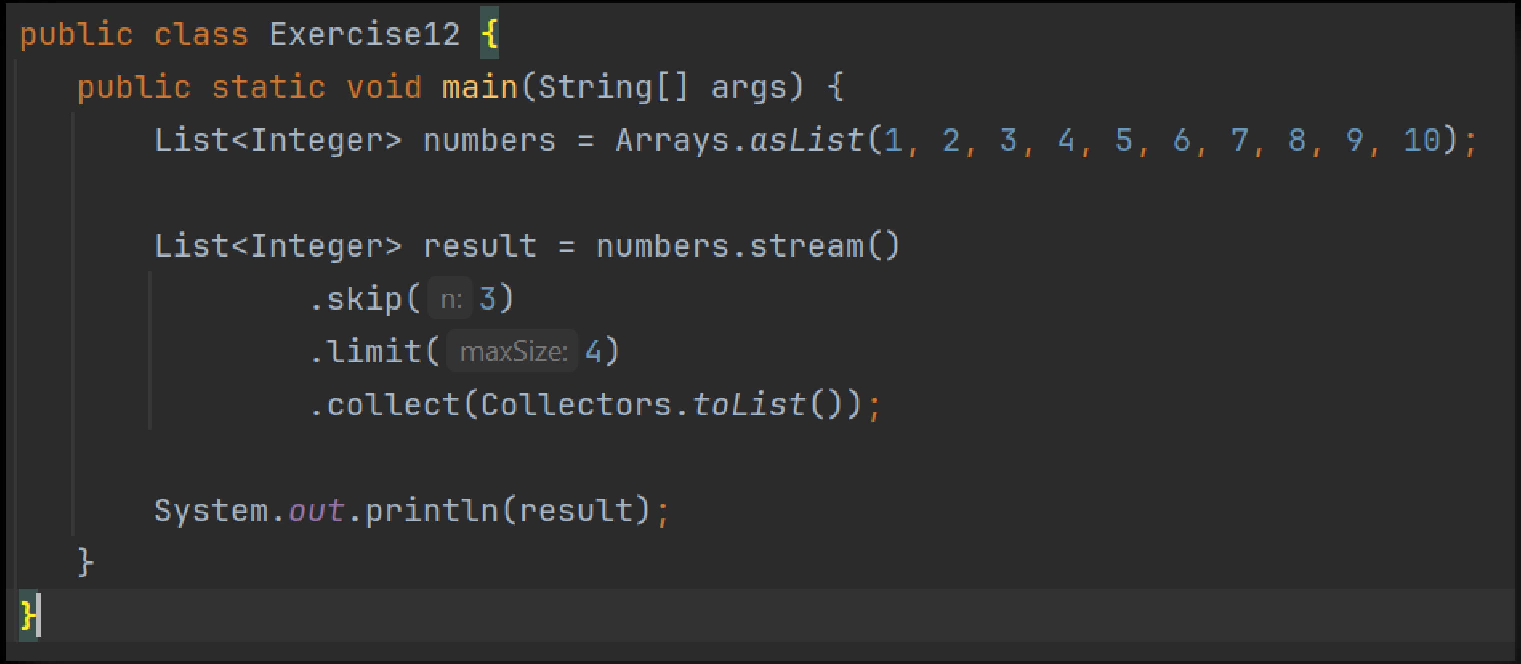


```java
public class Exercise11 {
    public static void main(String[] args) {
        List<String> list = List.of("one", "two", "three");

        list.add("four");
    }
}
```

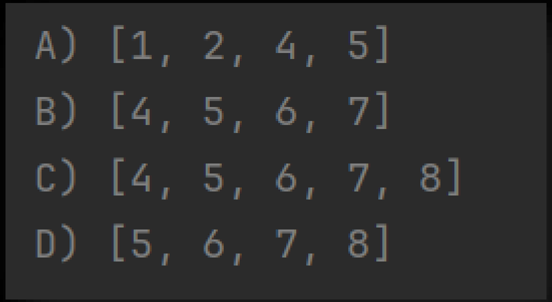


```
A) 'UnsupportedOperationException'
```

12: Combining skip and limit

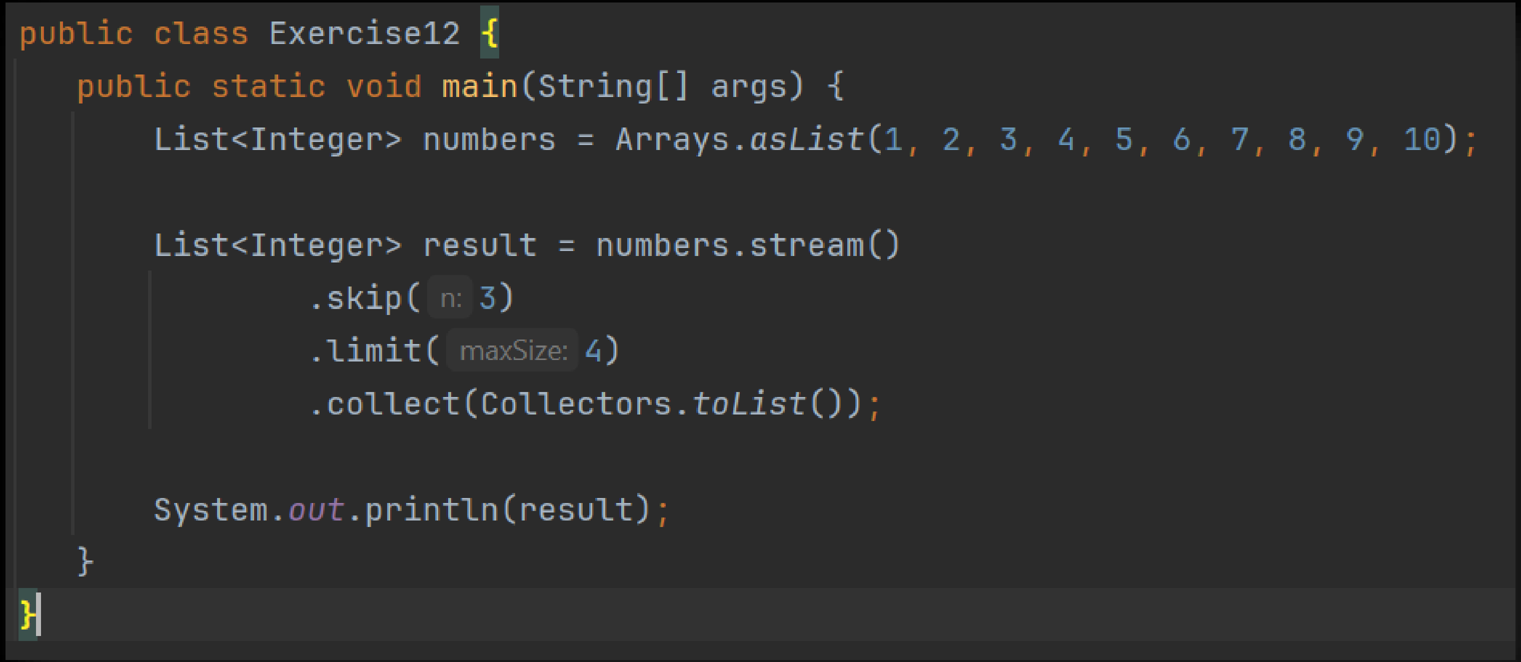


```
public class Exercise12 {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

        List<Integer> result = numbers.stream()
                .skip( n: 3)
                .limit( maxSize: 4)
                .collect(Collectors.toList());

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

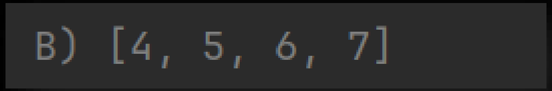


```
A) [1, 2, 4, 5]
B) [4, 5, 6, 7]
C) [4, 5, 6, 7, 8]
D) [5, 6, 7, 8]
```

12: Combining skip and limit

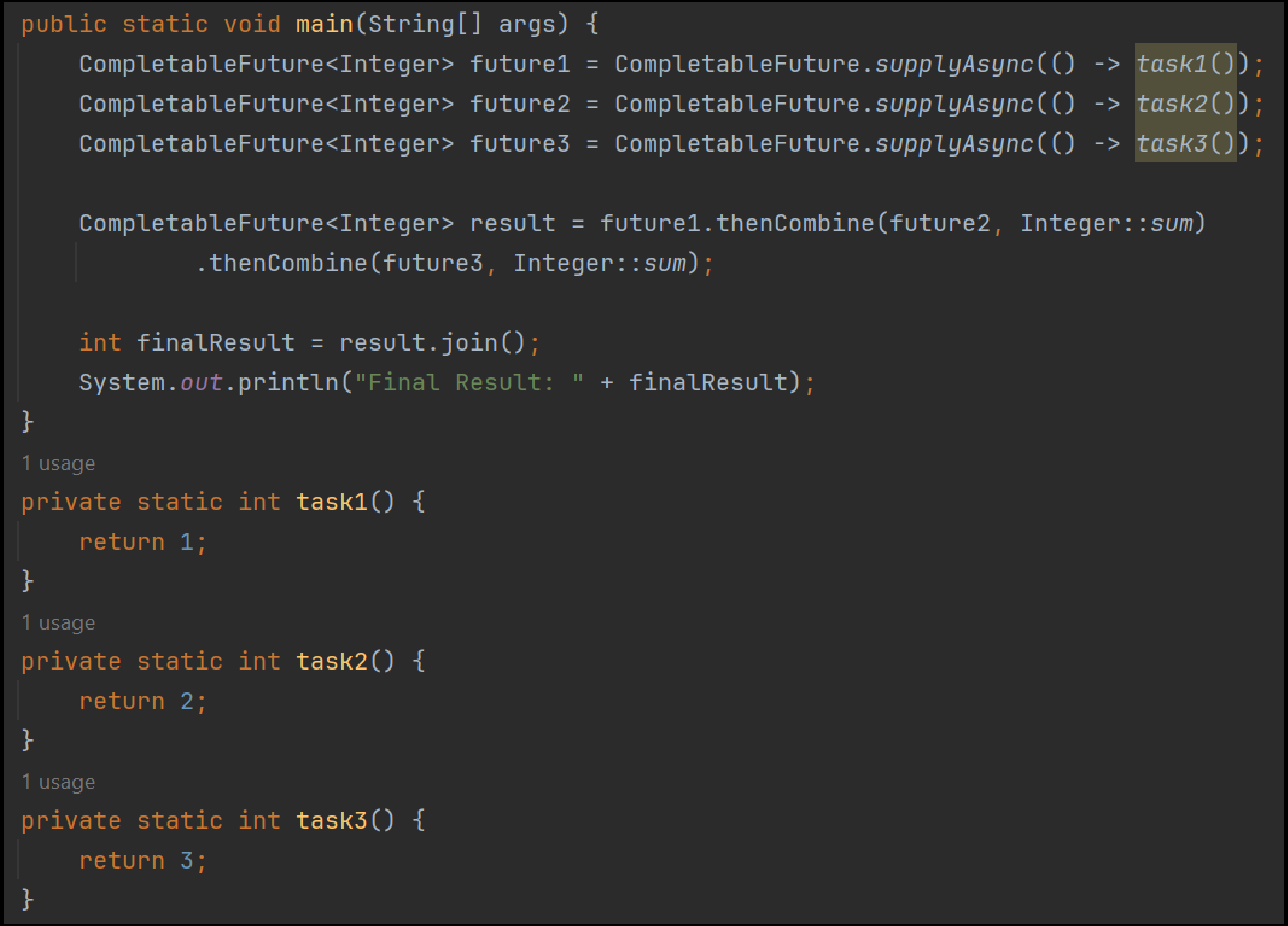


```java
public class Exercise12 {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

        List<Integer> result = numbers.stream()
                .skip( n: 3)
                .limit( maxSize: 4)
                .collect(Collectors.toList());

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



```
B) [4, 5, 6, 7]
```

13: Concurrency with CompletableFuture



```java
public static void main(String[] args) {
    CompletableFuture<Integer> future1 = CompletableFuture.supplyAsync(() -> task1());
    CompletableFuture<Integer> future2 = CompletableFuture.supplyAsync(() -> task2());
    CompletableFuture<Integer> future3 = CompletableFuture.supplyAsync(() -> task3());

    CompletableFuture<Integer> result = future1.thenCombine(future2, Integer::sum)
            .thenCombine(future3, Integer::sum);

    int finalResult = result.join();
    System.out.println("Final Result: " + finalResult);
}
1 usage
private static int task1() {
    return 1;
}
1 usage
private static int task2() {
    return 2;
}
1 usage
private static int task3() {
    return 3;
}
```

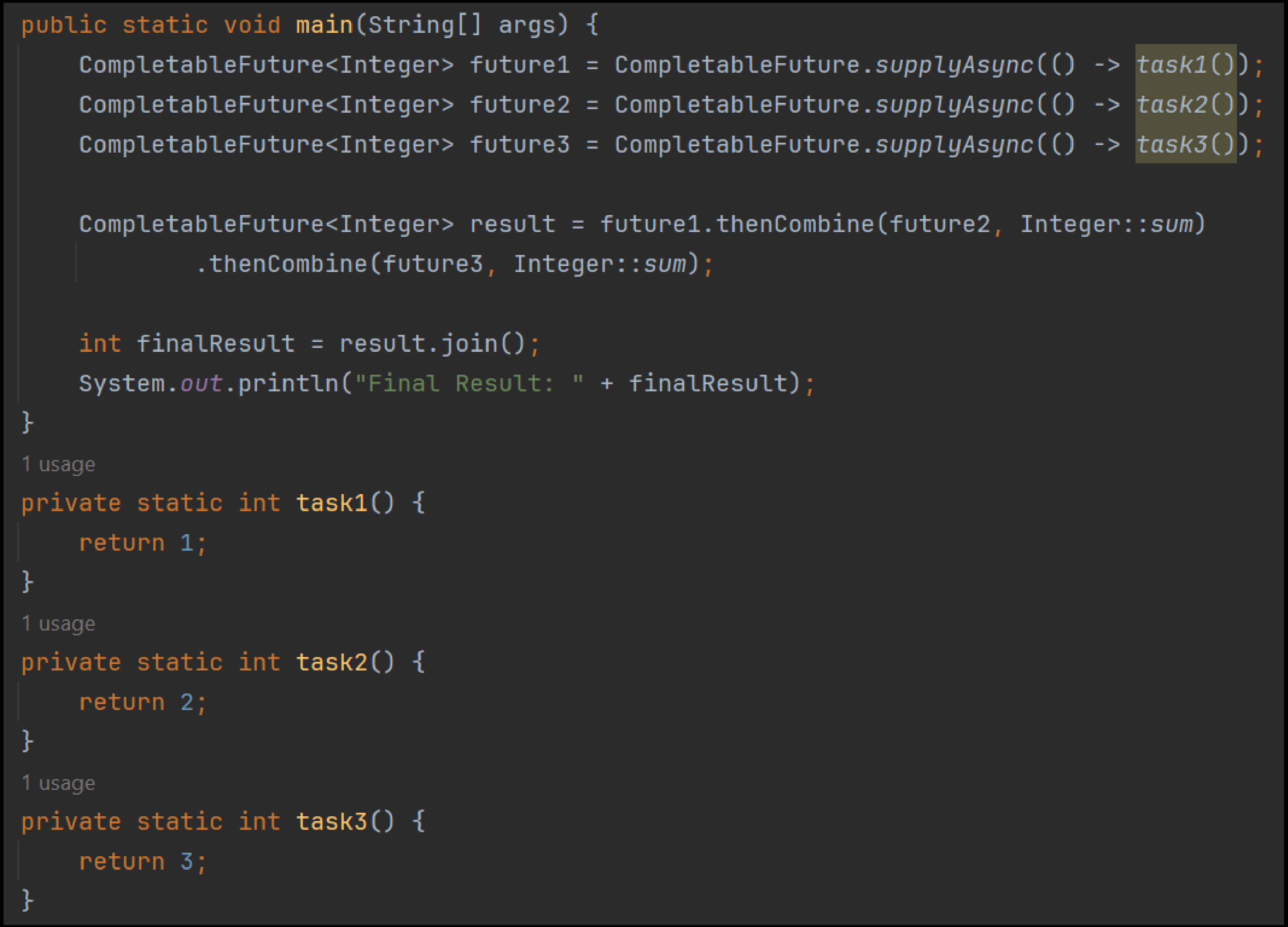
A) Replace thenCombine with allOf.

B) Add an explicit executor to supplyAsync.

C) Use thenCompose instead of thenCombine.

D) Change join to get.

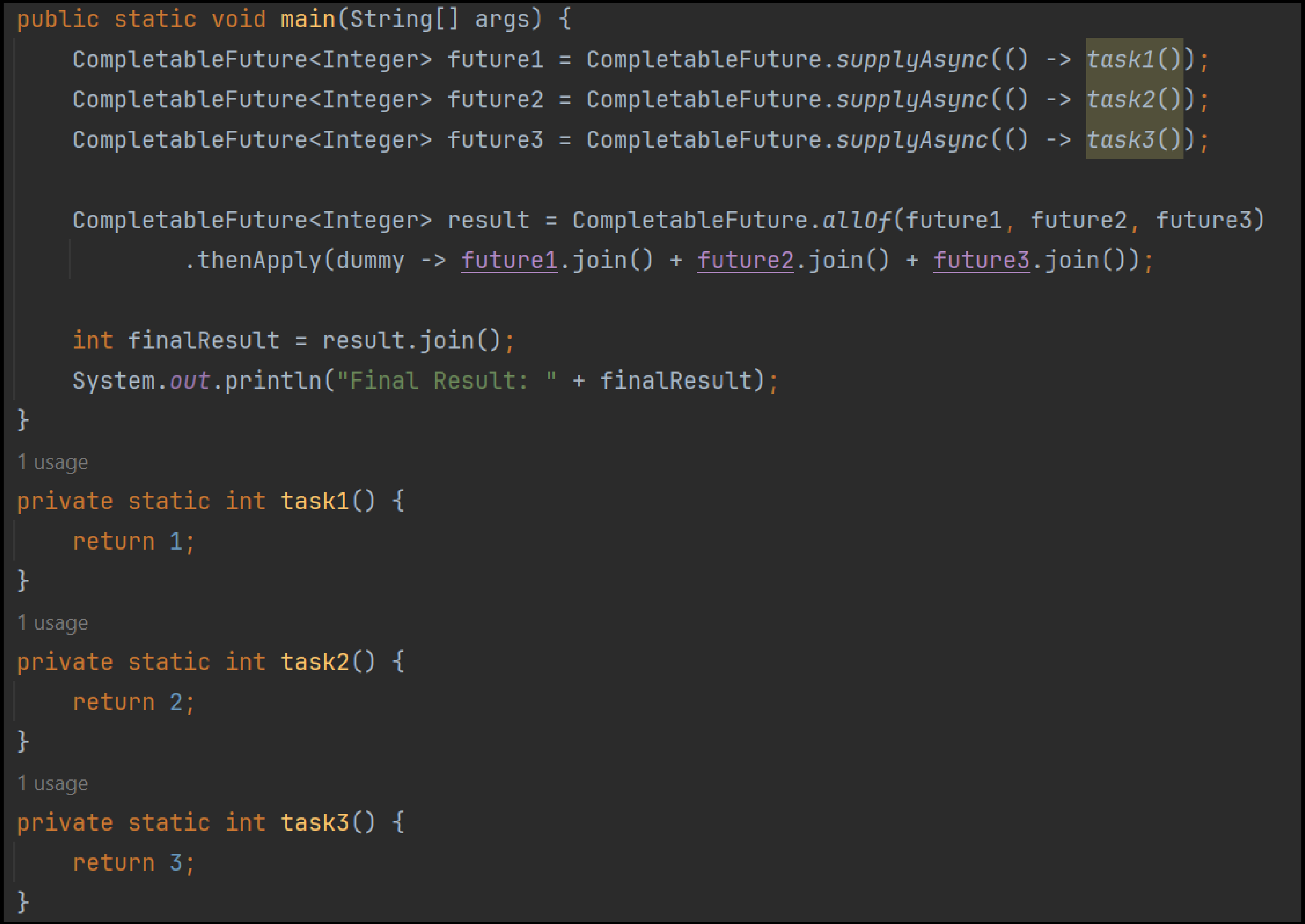
13: Concurrency with CompletableFuture



```java
public static void main(String[] args) {
    CompletableFuture<Integer> future1 = CompletableFuture.supplyAsync(() -> task1());
    CompletableFuture<Integer> future2 = CompletableFuture.supplyAsync(() -> task2());
    CompletableFuture<Integer> future3 = CompletableFuture.supplyAsync(() -> task3());

    CompletableFuture<Integer> result = future1.thenCombine(future2, Integer::sum)
            .thenCombine(future3, Integer::sum);

    int finalResult = result.join();
    System.out.println("Final Result: " + finalResult);
}
1 usage
private static int task1() {
    return 1;
}
1 usage
private static int task2() {
    return 2;
}
1 usage
private static int task3() {
    return 3;
}
```

```
A) Replace thenCombine with allOf.
```

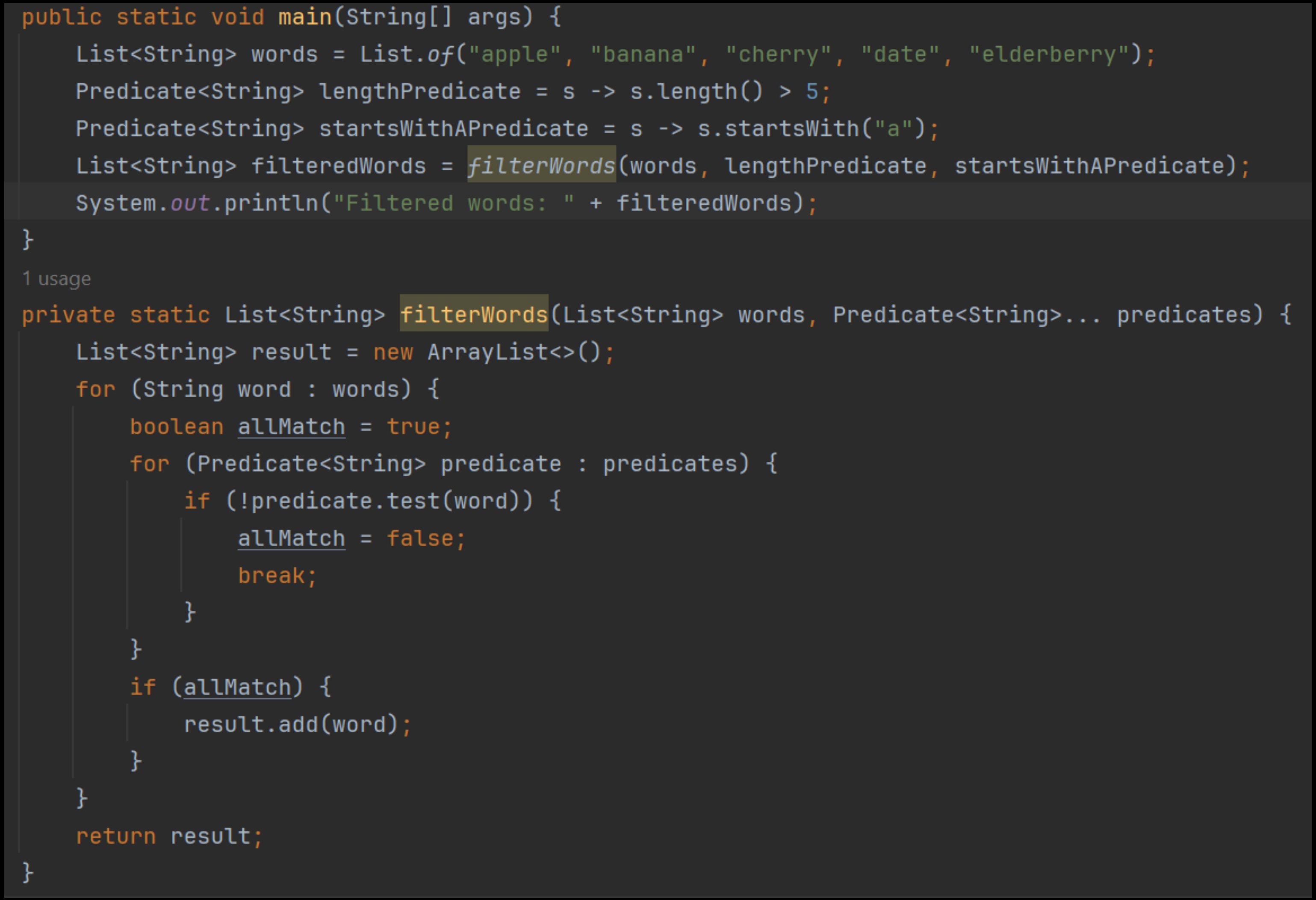
13: Concurrency with CompletableFuture



```
public static void main(String[] args) {
    CompletableFuture<Integer> future1 = CompletableFuture.supplyAsync(() -> task1());
    CompletableFuture<Integer> future2 = CompletableFuture.supplyAsync(() -> task2());
    CompletableFuture<Integer> future3 = CompletableFuture.supplyAsync(() -> task3());

    CompletableFuture<Integer> result = CompletableFuture.allOf(future1, future2, future3)
            .thenApply(dummy -> future1.join() + future2.join() + future3.join());

    int finalResult = result.join();
    System.out.println("Final Result: " + finalResult);
}
1 usage
private static int task1() {
    return 1;
}
1 usage
private static int task2() {
    return 2;
}
1 usage
private static int task3() {
    return 3;
}
```

A) Replace thenCombine with allOf.

14: Lambda Expressions



```
public static void main(String[] args) {
    List<String> words = List.of("apple", "banana", "cherry", "date", "elderberry");
    Predicate<String> lengthPredicate = s -> s.length() > 5;
    Predicate<String> startsWithAPredicate = s -> s.startsWith("a");
    List<String> filteredWords = filterWords(words, lengthPredicate, startsWithAPredicate);
    System.out.println("Filtered words: " + filteredWords);
}
1 usage
private static List<String> filterWords(List<String> words, Predicate<String>... predicates) {
    List<String> result = new ArrayList<>();
    for (String word : words) {
        boolean allMatch = true;
        for (Predicate<String> predicate : predicates) {
            if (!predicate.test(word)) {
                allMatch = false;
                break;
            }
        }
        if (allMatch) {
            result.add(word);
        }
    }
    return result;
}
```

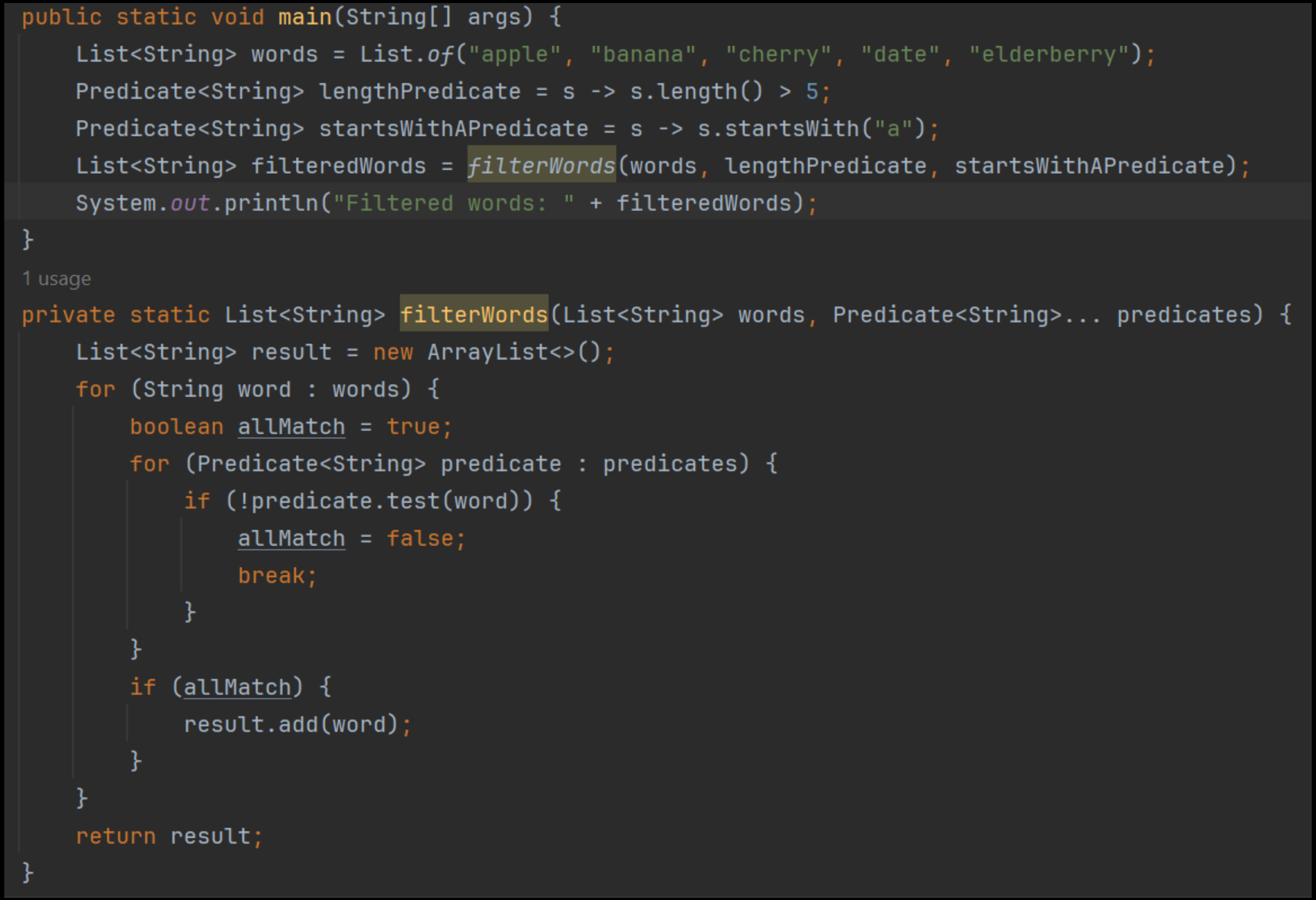
A) Refactor filterWords method to use Stream API and combine predicates using and.

B) Inline Predicate Combination in main.

C) Use filterWords method as is without any modifications.

D) Replace Predicate with custom functional interfaces for better performance.

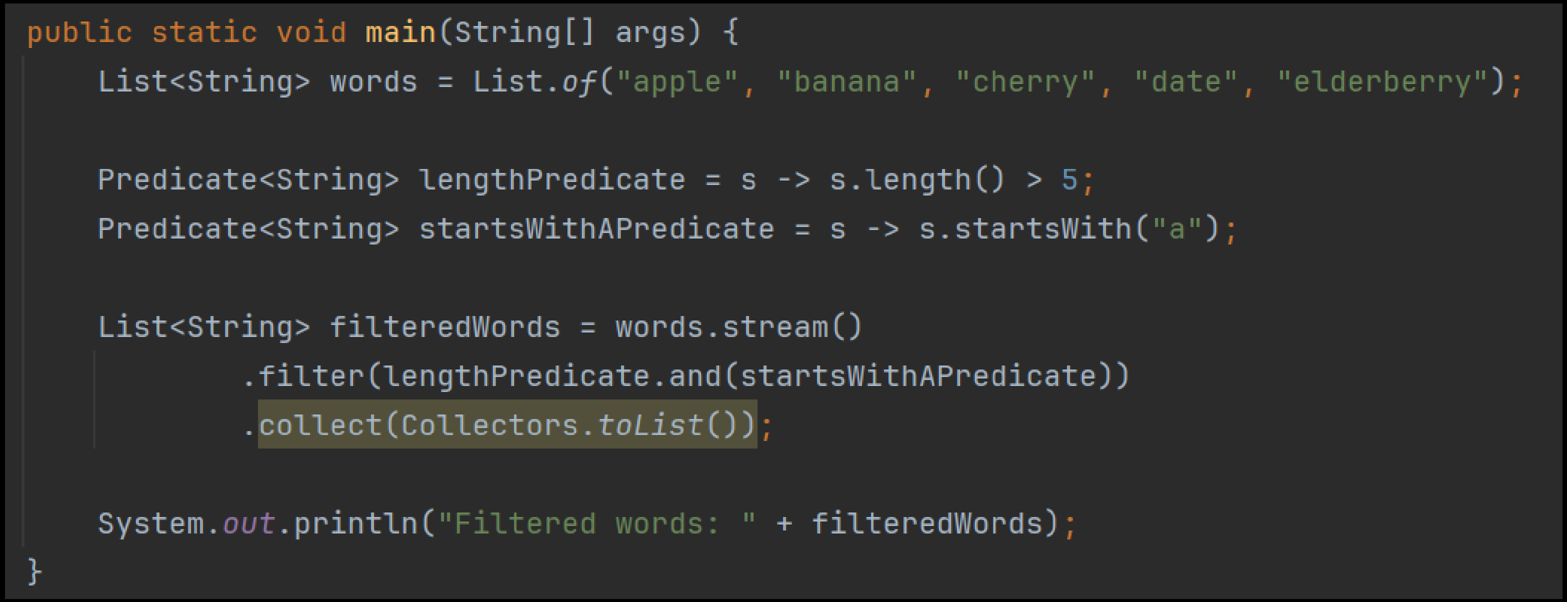
14: Lambda Expressions



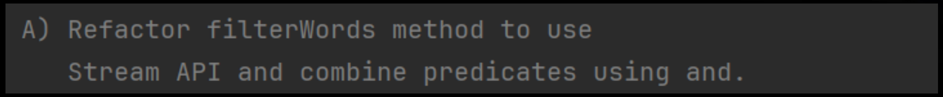
```java
public static void main(String[] args) {
    List<String> words = List.of("apple", "banana", "cherry", "date", "elderberry");
    Predicate<String> lengthPredicate = s -> s.length() > 5;
    Predicate<String> startsWithAPredicate = s -> s.startsWith("a");
    List<String> filteredWords = filterWords(words, lengthPredicate, startsWithAPredicate);
    System.out.println("Filtered words: " + filteredWords);
}
1 usage
private static List<String> filterWords(List<String> words, Predicate<String>... predicates) {
    List<String> result = new ArrayList<>();
    for (String word : words) {
        boolean allMatch = true;
        for (Predicate<String> predicate : predicates) {
            if (!predicate.test(word)) {
                allMatch = false;
                break;
            }
        }
        if (allMatch) {
            result.add(word);
        }
    }
    return result;
}
```

A) Refactor filterWords method to use Stream API and combine predicates using and.

14: Lambda Expressions

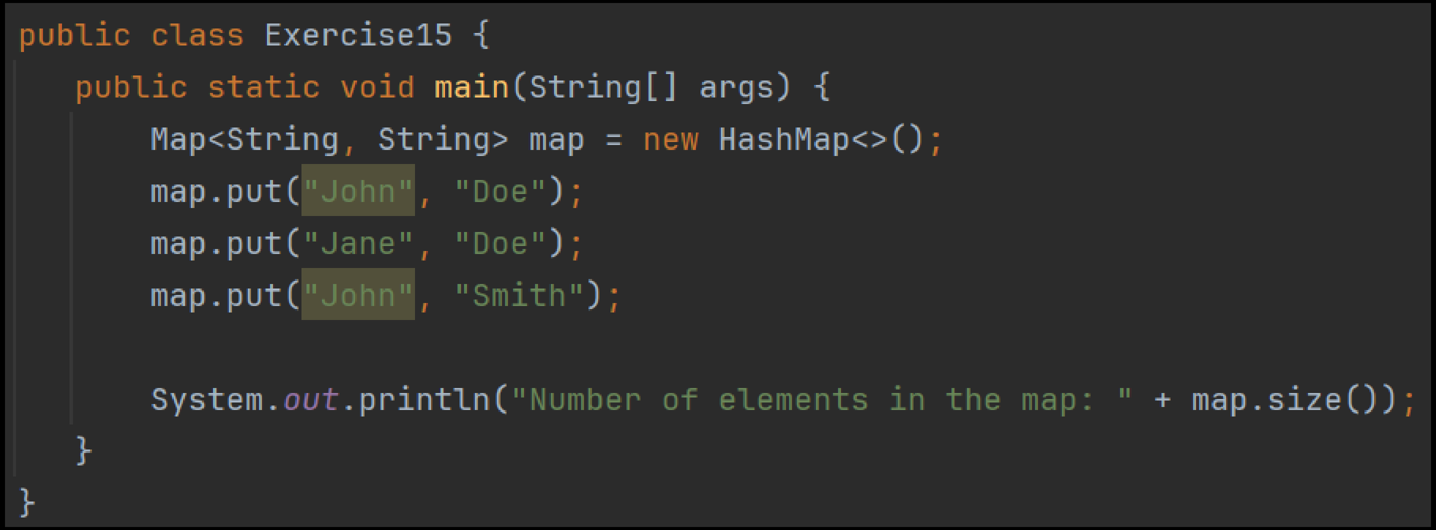


```java
public static void main(String[] args) {
    List<String> words = List.of("apple", "banana", "cherry", "date", "elderberry");

    Predicate<String> lengthPredicate = s -> s.length() > 5;
    Predicate<String> startsWithAPredicate = s -> s.startsWith("a");

    List<String> filteredWords = words.stream()
            .filter(lengthPredicate.and(startsWithAPredicate))
            .collect(Collectors.toList());

    System.out.println("Filtered words: " + filteredWords);
}
```

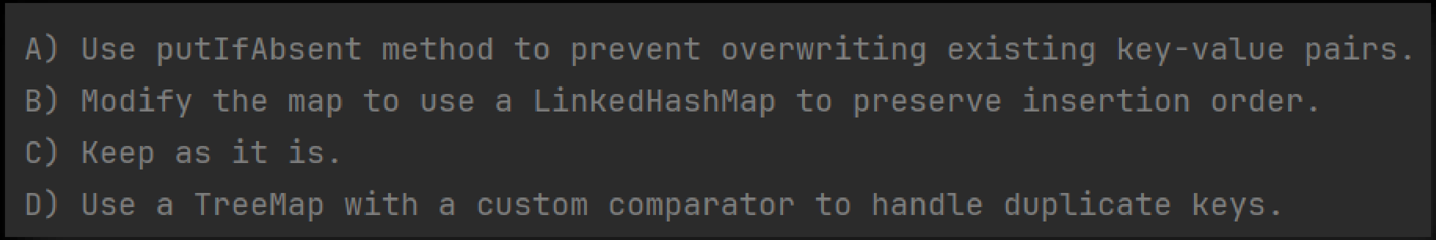


```
A) Refactor filterWords method to use
   Stream API and combine predicates using and.
```

15: Map

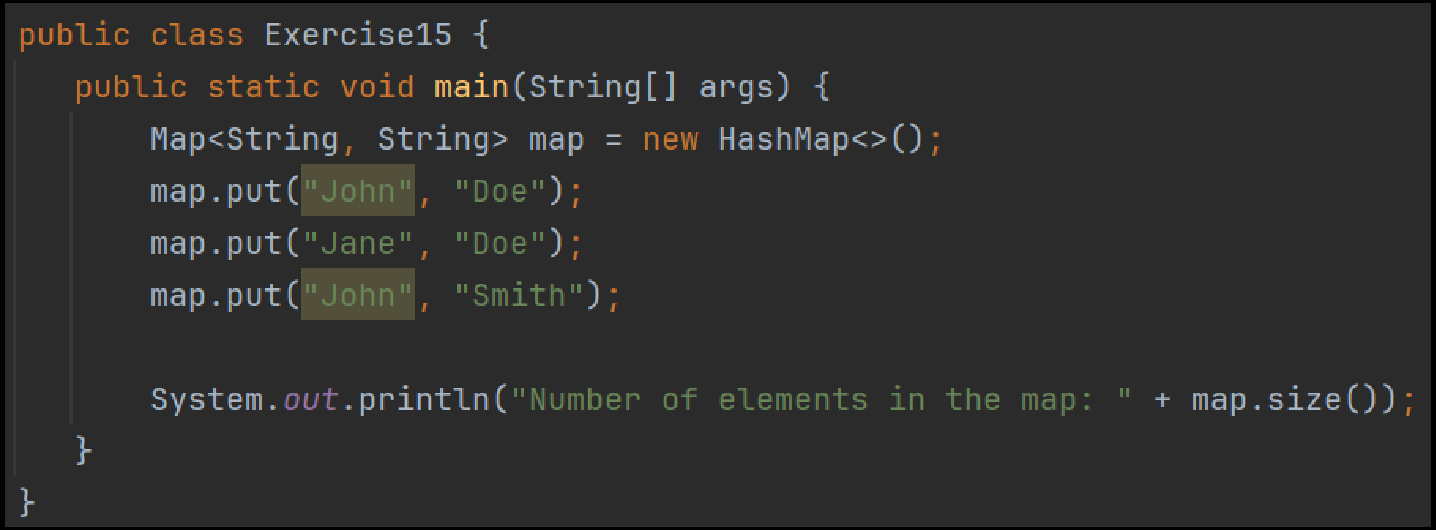


```
public class Exercise15 {
    public static void main(String[] args) {
        Map<String, String> map = new HashMap<>();
        map.put("John", "Doe");
        map.put("Jane", "Doe");
        map.put("John", "Smith");

        System.out.println("Number of elements in the map: " + map.size());
    }
}
```



```
A) Use putIfAbsent method to prevent overwriting existing key-value pairs.
B) Modify the map to use a LinkedHashMap to preserve insertion order.
C) Keep as it is.
D) Use a TreeMap with a custom comparator to handle duplicate keys.
```

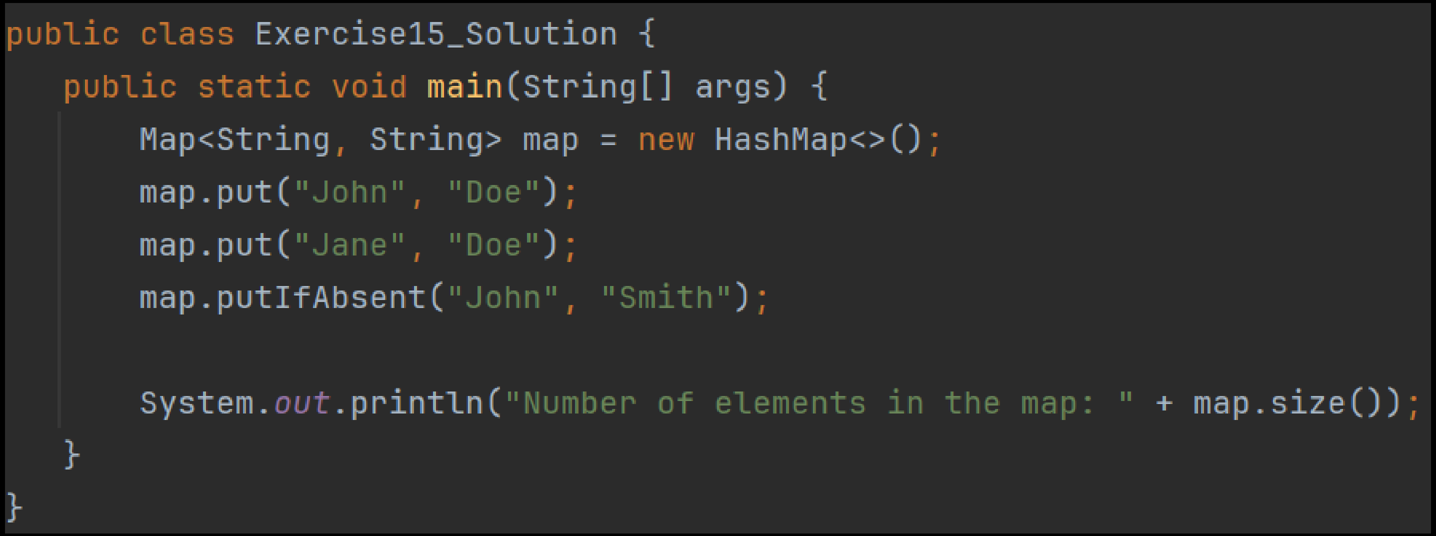
15: Map



```java
public class Exercise15 {
    public static void main(String[] args) {
        Map<String, String> map = new HashMap<>();
        map.put("John", "Doe");
        map.put("Jane", "Doe");
        map.put("John", "Smith");

        System.out.println("Number of elements in the map: " + map.size());
    }
}
```

A) Use putIfAbsent method to prevent overwriting existing key-value pairs.

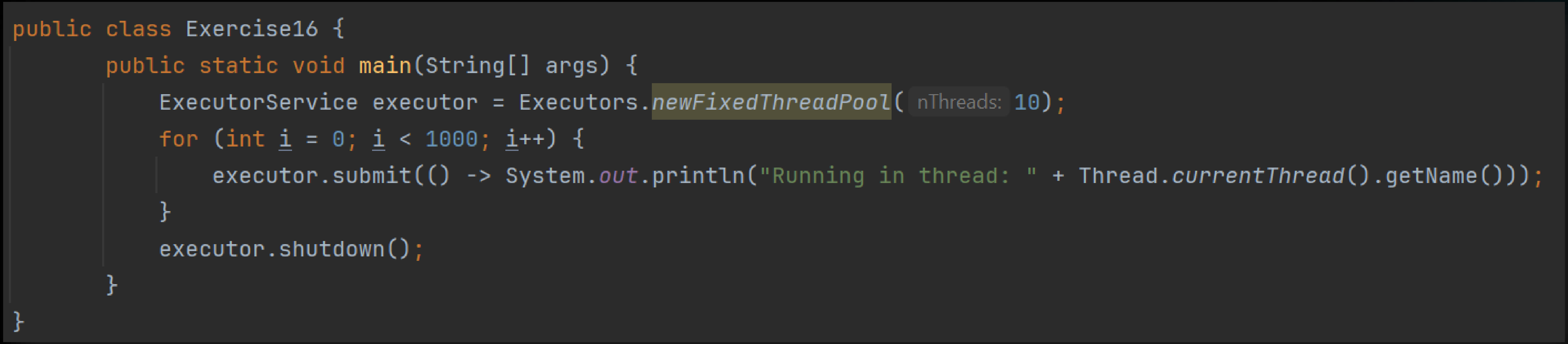
15: Map



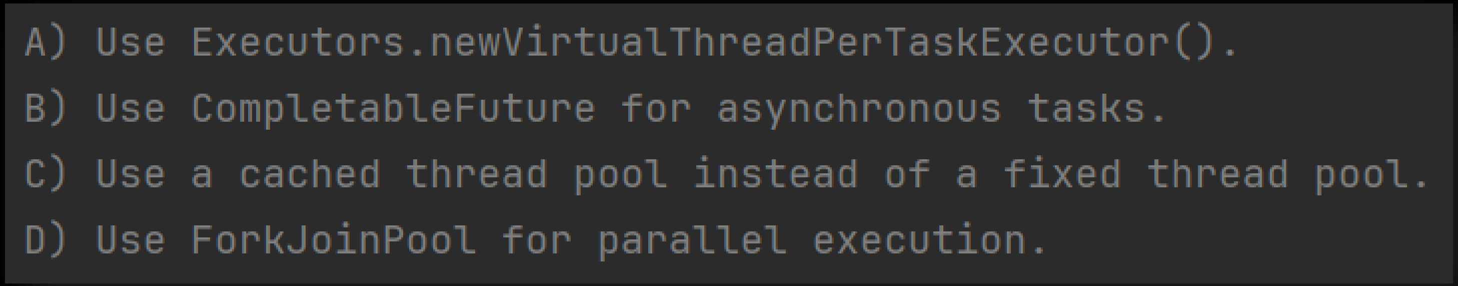
```java
public class Exercise15_Solution {
    public static void main(String[] args) {
        Map<String, String> map = new HashMap<>();
        map.put("John", "Doe");
        map.put("Jane", "Doe");
        map.putIfAbsent("John", "Smith");

        System.out.println("Number of elements in the map: " + map.size());
    }
}
```

A) Use putIfAbsent method to prevent overwriting existing key-value pairs.

16: Virtual Threads

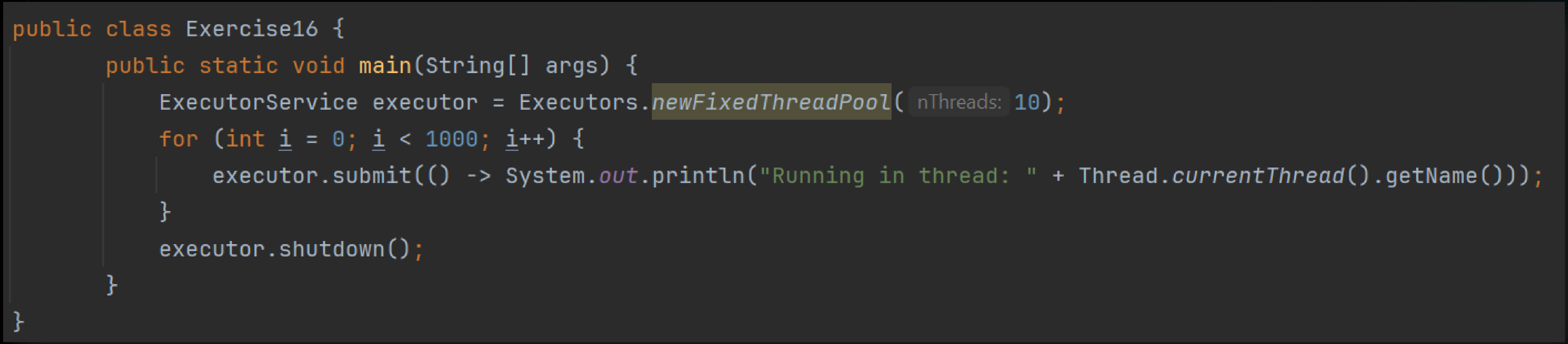


```
public class Exercise16 {
        public static void main(String[] args) {
            ExecutorService executor = Executors.newFixedThreadPool( nThreads: 10);
            for (int i = 0; i < 1000; i++) {
                executor.submit(() -> System.out.println("Running in thread: " + Thread.currentThread().getName()));
            }
            executor.shutdown();
        }
}
```



```
A) Use Executors.newVirtualThreadPerTaskExecutor().
B) Use CompletableFuture for asynchronous tasks.
C) Use a cached thread pool instead of a fixed thread pool.
D) Use ForkJoinPool for parallel execution.
```

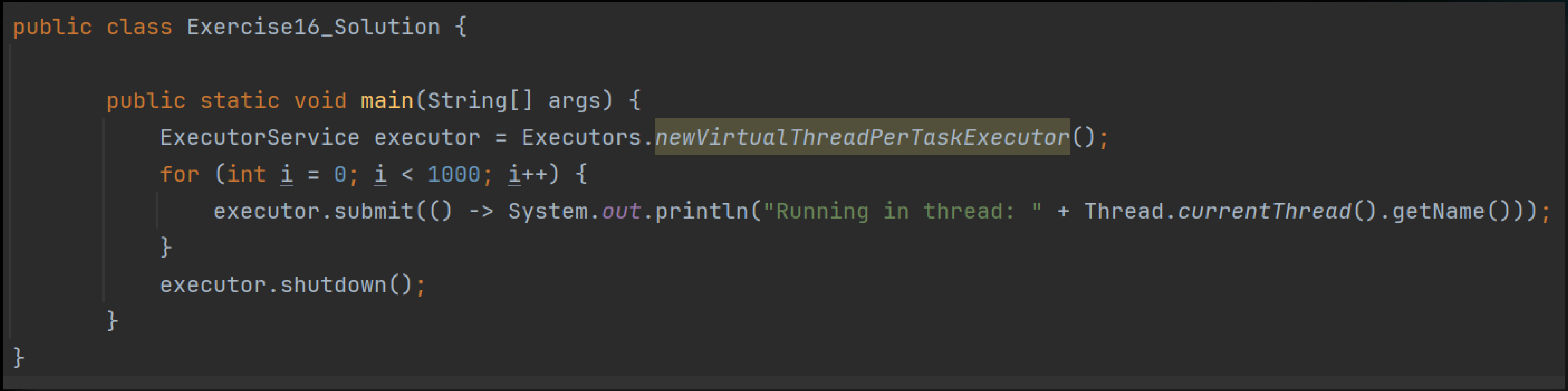
16: Virtual Threads



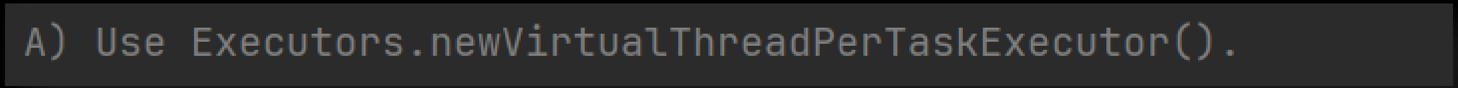
```java
public class Exercise16 {
        public static void main(String[] args) {
            ExecutorService executor = Executors.newFixedThreadPool( nThreads: 10);
            for (int i = 0; i < 1000; i++) {
                executor.submit(() -> System.out.println("Running in thread: " + Thread.currentThread().getName()));
            }
            executor.shutdown();
        }
}
```

A) Use Executors.newVirtualThreadPerTaskExecutor().

16: Virtual Threads

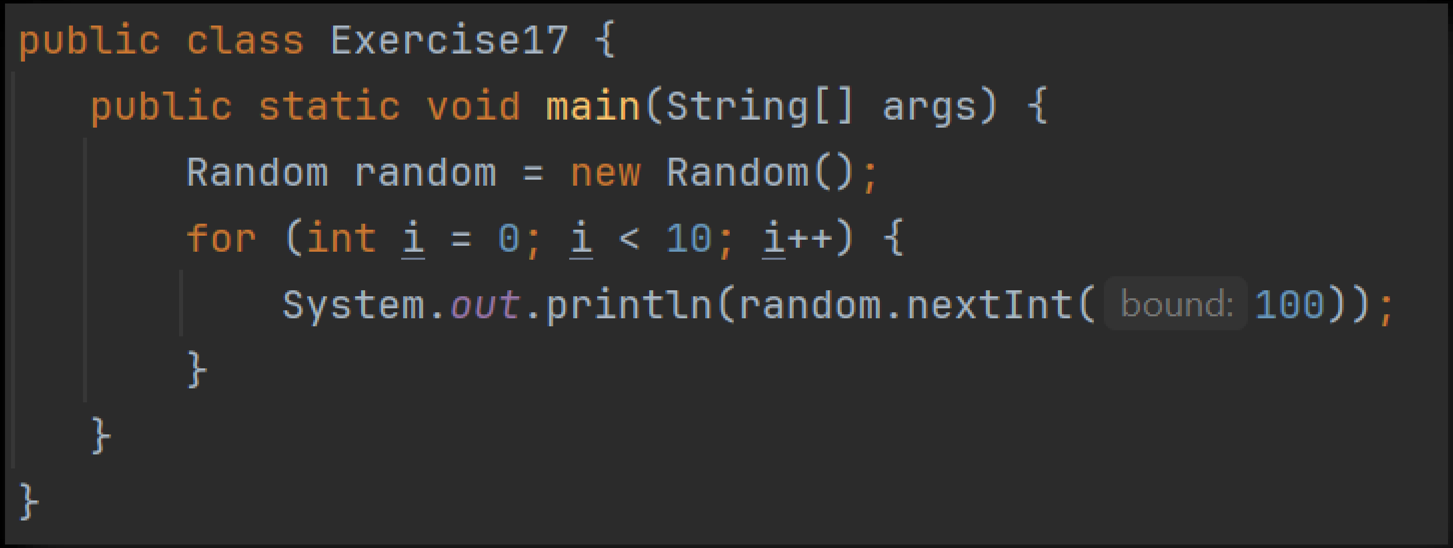


```java
public class Exercise16_Solution {

        public static void main(String[] args) {
            ExecutorService executor = Executors.newVirtualThreadPerTaskExecutor();
            for (int i = 0; i < 1000; i++) {
                executor.submit(() -> System.out.println("Running in thread: " + Thread.currentThread().getName()));
            }
            executor.shutdown();
        }
}
```

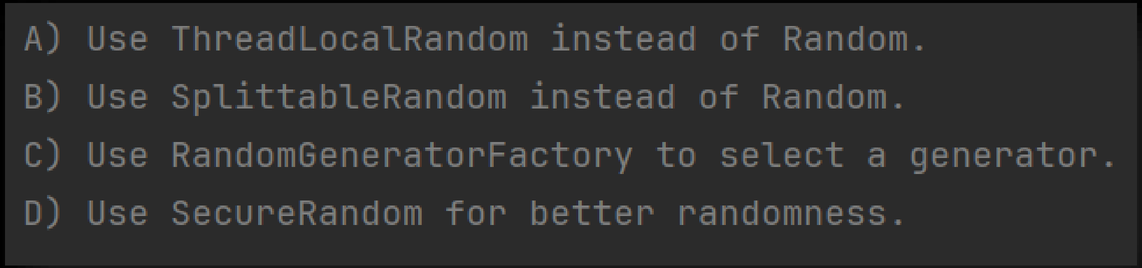


A) Use Executors.newVirtualThreadPerTaskExecutor().

17: Randomness

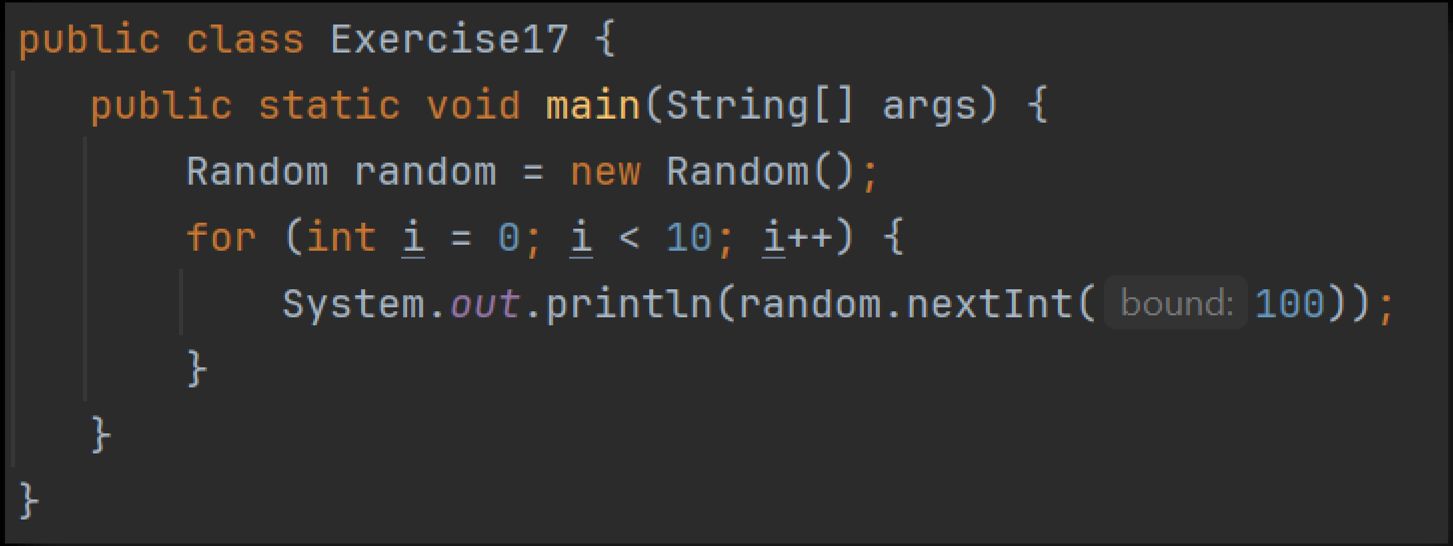


```java
public class Exercise17 {
    public static void main(String[] args) {
        Random random = new Random();
        for (int i = 0; i < 10; i++) {
            System.out.println(random.nextInt( bound: 100));
        }
    }
}
```

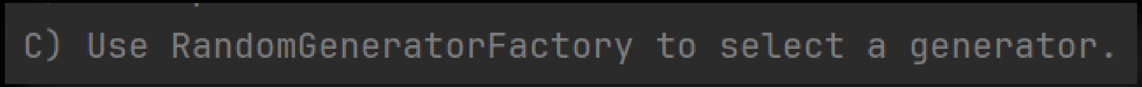


```
A) Use ThreadLocalRandom instead of Random.
B) Use SplittableRandom instead of Random.
C) Use RandomGeneratorFactory to select a generator.
D) Use SecureRandom for better randomness.
```

17: Randomness

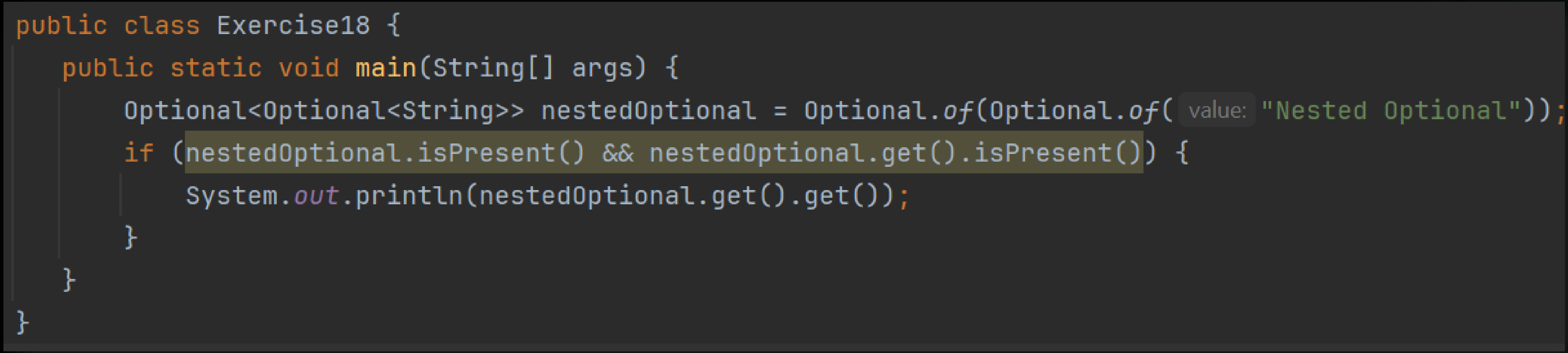


```
public class Exercise17 {
    public static void main(String[] args) {
        Random random = new Random();
        for (int i = 0; i < 10; i++) {
            System.out.println(random.nextInt( bound: 100));
        }
    }
}
```

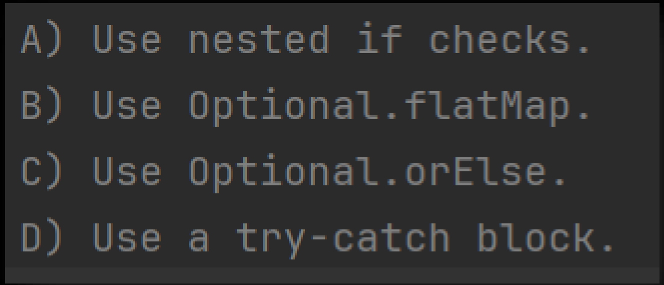


```
C) Use RandomGeneratorFactory to select a generator.
```

18: Optional

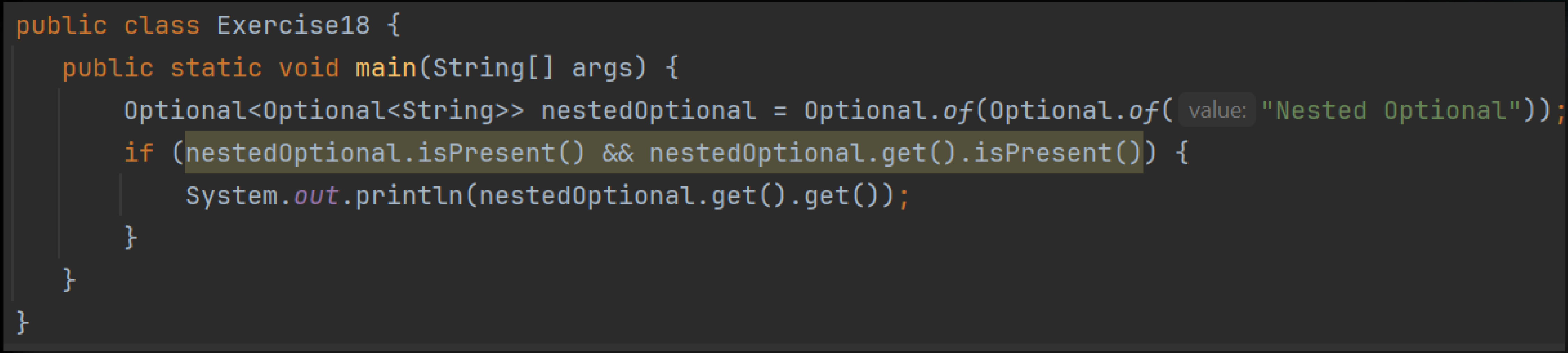


```java
public class Exercise18 {
    public static void main(String[] args) {
        Optional<Optional<String>> nestedOptional = Optional.of(Optional.of( value: "Nested Optional"));
        if (nestedOptional.isPresent() && nestedOptional.get().isPresent()) {
            System.out.println(nestedOptional.get().get());
        }
    }
}
```

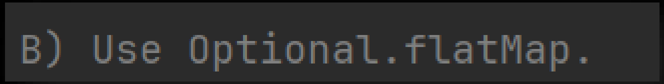


```
A) Use nested if checks.
B) Use Optional.flatMap.
C) Use Optional.orElse.
D) Use a try-catch block.
```

18: Optional

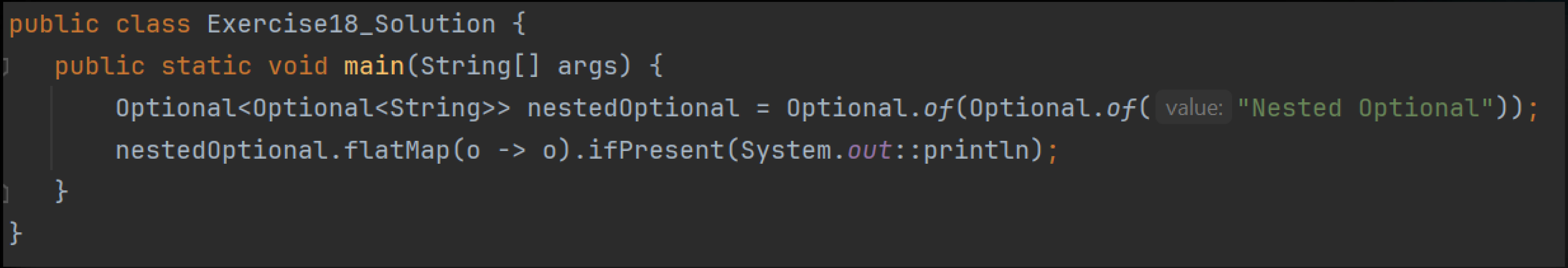


```java
public class Exercise18 {
    public static void main(String[] args) {
        Optional<Optional<String>> nestedOptional = Optional.of(Optional.of( value: "Nested Optional"));
        if (nestedOptional.isPresent() && nestedOptional.get().isPresent()) {
            System.out.println(nestedOptional.get().get());
        }
    }
}
```

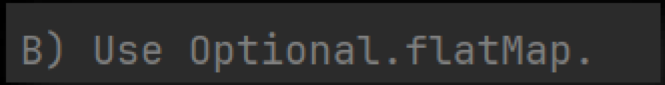


B) Use Optional.flatMap.

18: Optional

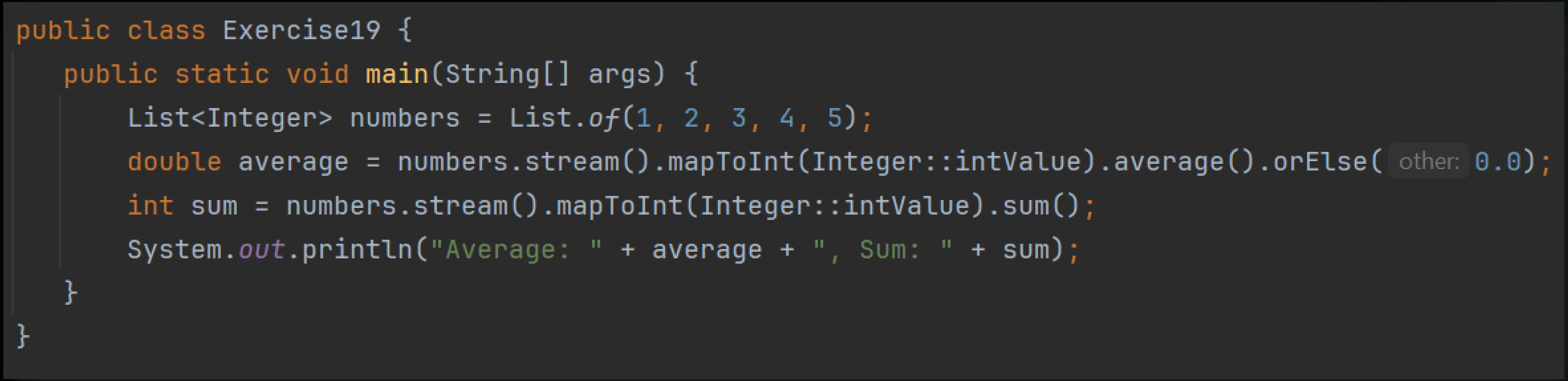


```
public class Exercise18_Solution {
    public static void main(String[] args) {
        Optional<Optional<String>> nestedOptional = Optional.of(Optional.of( value: "Nested Optional"));
        nestedOptional.flatMap(o -> o).ifPresent(System.out::println);
    }
}
```

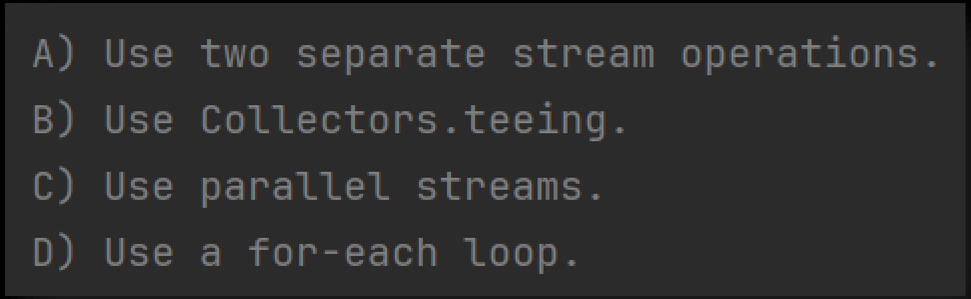


```
B) Use Optional.flatMap.
```

19: Streams

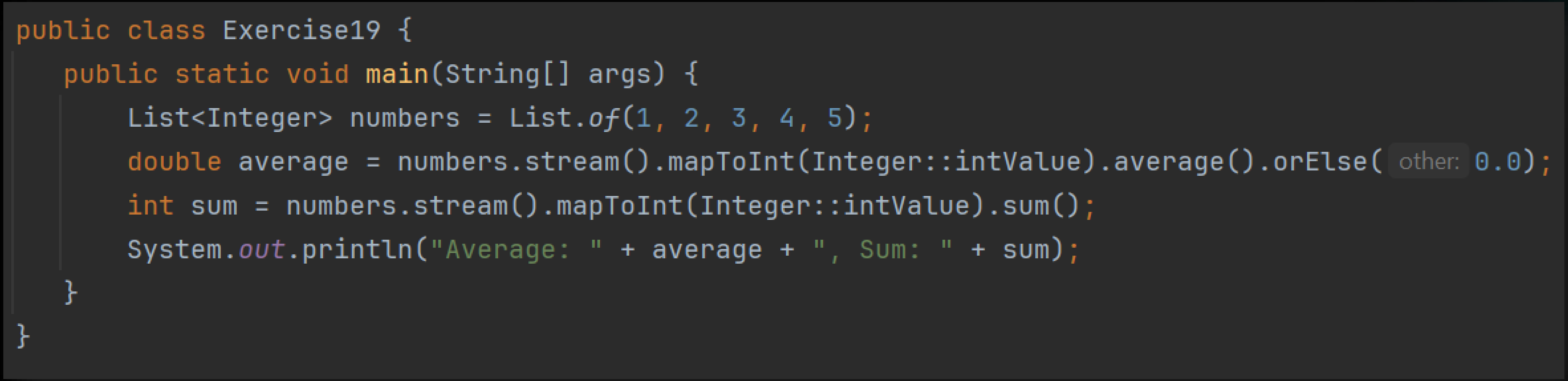


```java
public class Exercise19 {
    public static void main(String[] args) {
        List<Integer> numbers = List.of(1, 2, 3, 4, 5);
        double average = numbers.stream().mapToInt(Integer::intValue).average().orElse( other: 0.0);
        int sum = numbers.stream().mapToInt(Integer::intValue).sum();
        System.out.println("Average: " + average + ", Sum: " + sum);
    }
}
```

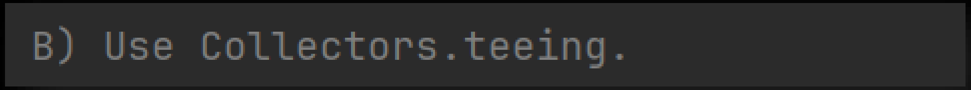


```
A) Use two separate stream operations.
B) Use Collectors.teeing.
C) Use parallel streams.
D) Use a for-each loop.
```

19: Streams

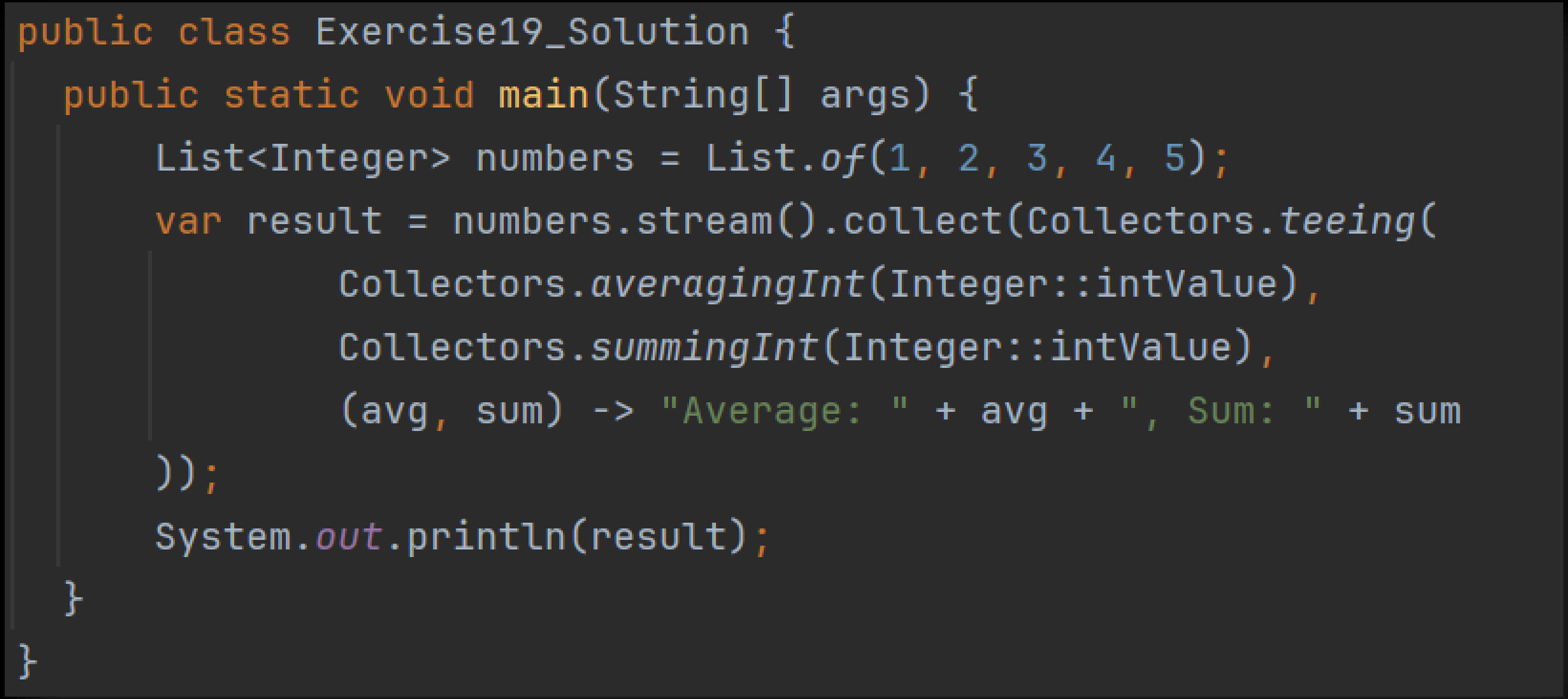


```
public class Exercise19 {
    public static void main(String[] args) {
        List<Integer> numbers = List.of(1, 2, 3, 4, 5);
        double average = numbers.stream().mapToInt(Integer::intValue).average().orElse( other: 0.0);
        int sum = numbers.stream().mapToInt(Integer::intValue).sum();
        System.out.println("Average: " + average + ", Sum: " + sum);
    }
}
```

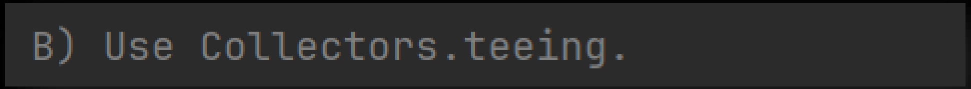


```
B) Use Collectors.teeing.
```

19: Streams

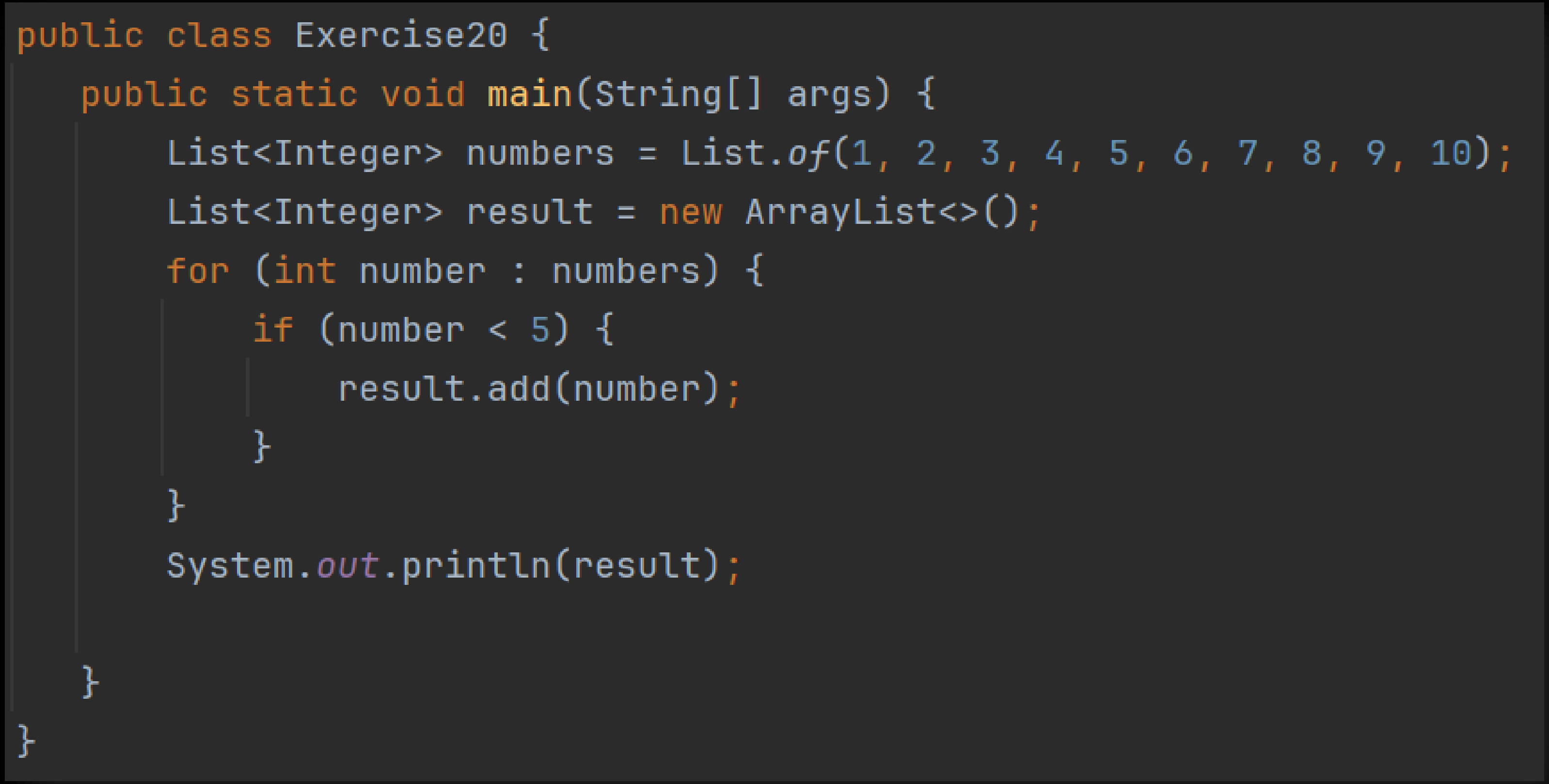


```
public class Exercise19_Solution {
  public static void main(String[] args) {
      List<Integer> numbers = List.of(1, 2, 3, 4, 5);
      var result = numbers.stream().collect(Collectors.teeing(
              Collectors.averagingInt(Integer::intValue),
              Collectors.summingInt(Integer::intValue),
              (avg, sum) -> "Average: " + avg + ", Sum: " + sum
      ));
      System.out.println(result);
  }
}
```

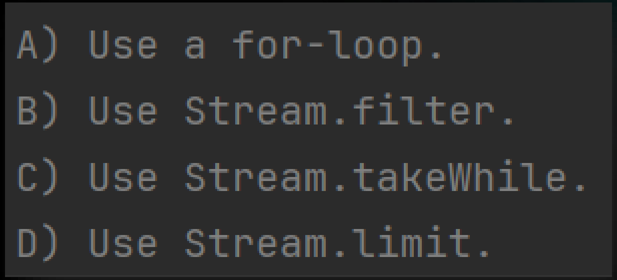


B) Use Collectors.teeing.

20: Streams

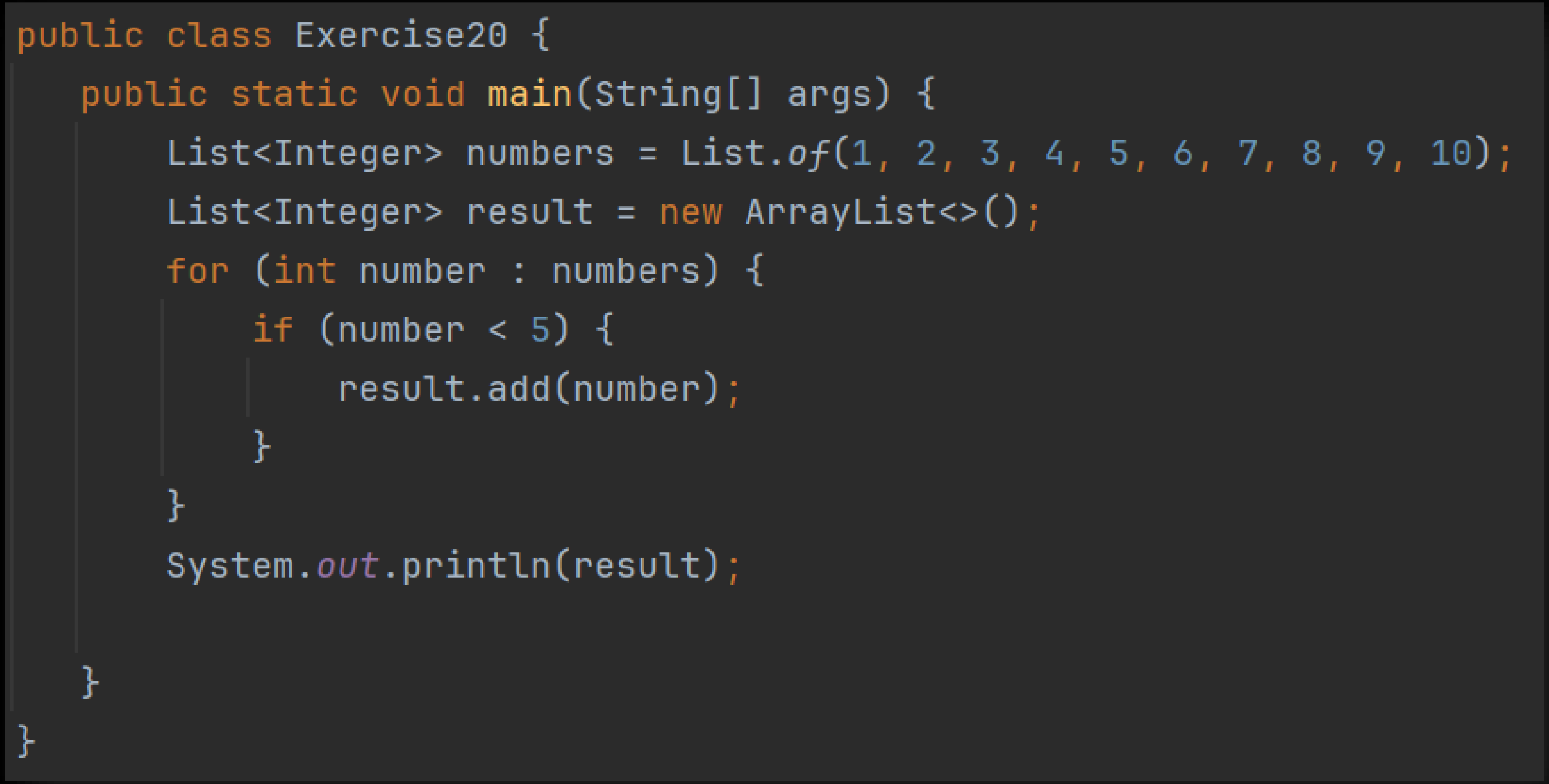


```java
public class Exercise20 {
    public static void main(String[] args) {
        List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
        List<Integer> result = new ArrayList<>();
        for (int number : numbers) {
            if (number < 5) {
                result.add(number);
            }
        }
        System.out.println(result);

    }
}
```

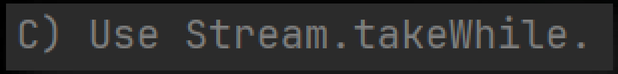


```
A) Use a for-loop.
B) Use Stream.filter.
C) Use Stream.takeWhile.
D) Use Stream.limit.
```

20: Streams

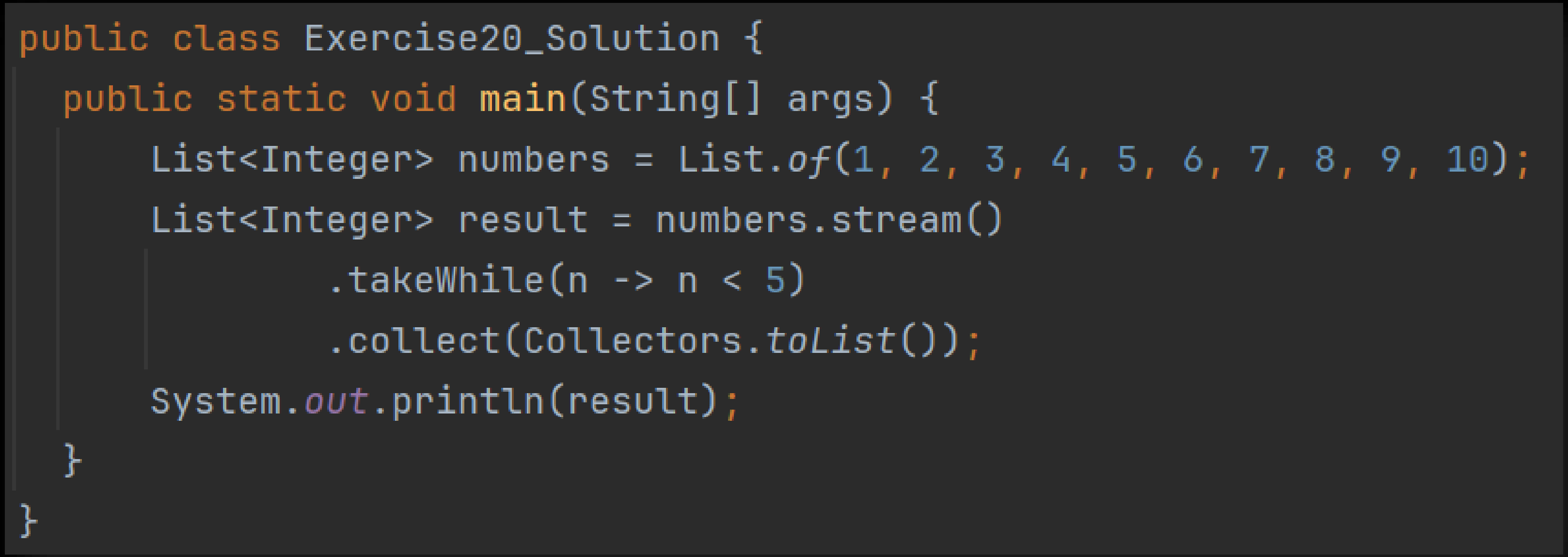


```java
public class Exercise20 {
    public static void main(String[] args) {
        List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
        List<Integer> result = new ArrayList<>();
        for (int number : numbers) {
            if (number < 5) {
                result.add(number);
            }
        }
        System.out.println(result);

    }
}
```

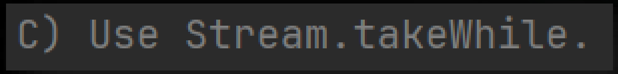


```
C) Use Stream.takeWhile.
```

20: Streams



```java
public class Exercise20_Solution {
  public static void main(String[] args) {
      List<Integer> numbers = List.of(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
      List<Integer> result = numbers.stream()
              .takeWhile(n -> n < 5)
              .collect(Collectors.toList());
      System.out.println(result);
  }
}
```



```
C) Use Stream.takeWhile.
```

Unlocking JAVA’s code maze

Thank you!