2025 — 2026

1TI – 1ACS



Back-End Development

Java Collections, Optionals & Debugging

N. Jacobs, J. Pieck, E. Steegmans, B. Van Impe, S. Van Peborgh, V. Witters

CONTENTS

- Collections
  - What is the collection framework in Java?
  - List and ArrayList
  - Set and HashSet
  - Map and HashMap
- Optionals
- Debugging
  - Stack trace
  - Debugging

Collections

Optionals

Debugging

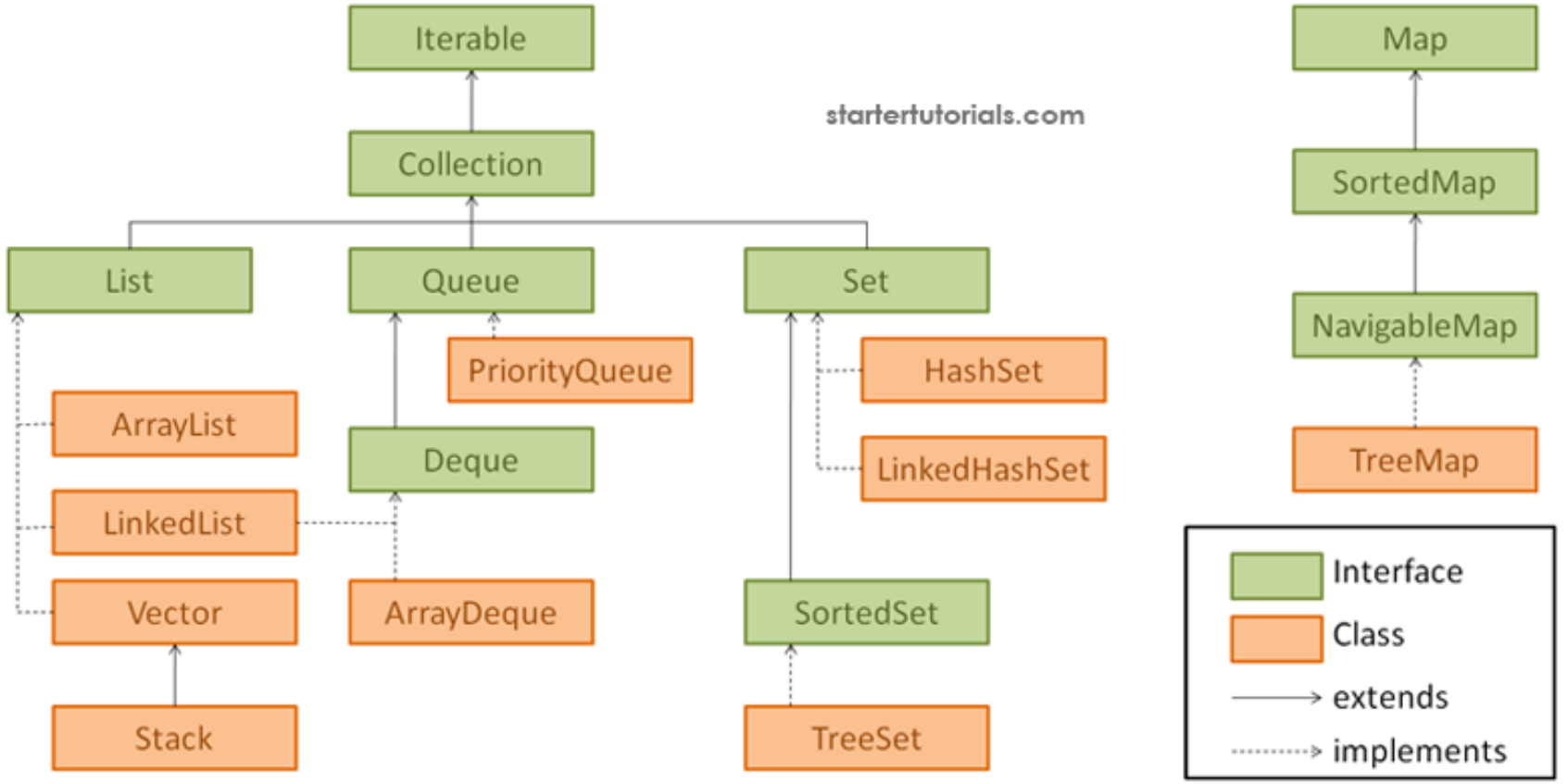
Collections

- What is the collection framework in Java?
- List and ArrayList
- Set and HashSet
- Map and HashMap

THE COLLECTION FRAMEWORK

- An array is a simple solution for grouping a fixed number of variables
- The collection framework is a set of built-in classes and interfaces in Java that provide a robust way to work with groups of data
  - The framework consists of interfaces that define a particular type of container
  - And classes that provide different implementations for the interfaces, each with their advantages and disadvantages in terms of performance, storage, …
- Basic interfaces are *Iterable* and *Collection*
- We will check out some interesting elements without diving too deeply into the classes

THE COLLECTION FRAMEWORK



LIST AND ARRAYLIST

```
public static void main(String[] args) {
    List<Integer> numbers = new ArrayList<Integer>();
    numbers.add(1);
    numbers.add(27);

    for (int number : numbers) {
        System.out.println(number);
    }
}
```

- A list is a collection of data of the same type with a fixed order and a dynamic size
- List is the interface
- ArrayList is the most used class implementing this interface
- For details, see previous class!

SET AND HASHSET

```
public static void main(String[] args) {
    Set<String> names = new HashSet<>();

    names.add("Bram");
    names.add("Roel");
    names.add("Emma");
    System.out.println(names.size());

    names.add("Roel");
    System.out.println(names.size());

    for (String name : names) {
        System.out.println(name);
    }
}
```

- A set is a collection of data of the same type where the order does not matter, and each element may only appear once in the set
- Set is the interface
- HashSet is the most used class implementing the interface

SET AND HASHSET

```
public static void main(String[] args) {
    Set<String> names = new HashSet<>();

    names.add("Bram");
    names.add("Roel");
    names.add("Emma");
    System.out.println(names.size());

    names.add("Roel");
    System.out.println(names.size());

    for (String name : names) {
        System.out.println(name);
    }
}
```

- Sets also work with generics
- HashSets are objects too and not primitive types

MAP AND HASHMAP

```
public static void main(String[] args) {
    Map<String, Integer> ages = new HashMap<>();

    ages.put("Bram", 33);
    ages.put("Roel", 46);
    ages.put("Emma", 72);
    System.out.println(ages.size());

    ages.put("Roel", 34);
    System.out.println(ages.size());
}
```

- A map is a storage place for "key, value" pairs
- Each key may occur only once
- Key and value may have different types
- Map is the interface
- HashMap is the most commonly used class implementing the interface

MAP AND HASHMAP

```
public static void main(String[] args) {
    Map<String, Integer> ages = new HashMap<>();

    // ...

    for (Map.Entry<String, Integer> entry : ages.entrySet()) {
        System.out.println(entry.getKey());
        System.out.println(ages.get(entry.getKey()));
    }

    for (String key : ages.keySet()) {
        System.out.println(key);
        System.out.println(ages.get(key));
    }

    for (Integer age : ages.values()) {
        System.out.println(age);
    }
}
```

- You can ask a map for the keys, values and entries
- Most of the time you are going to query a map primarily using the keys

EQUALS AND HASHCODE

```
public static void main(String[] args) {
    Set<String> names = new HashSet<>();

    names.add("Bram");
    names.add("Roel");
    names.add("Emma");
    System.out.println(names.size());

    names.add("Roel");
    System.out.println(names.size());

    for (String name : names) {
        System.out.println(name);
    }
}
```

- To check whether an object already exists in the set, the "equals" and "hashCode" methods are used
  - "equals" checks whether two objects are functionally equal
  - "hashCode" hashes the contents of the object to quickly check if they are approximately equal
- If you write your own object that will be used in a Set, it is best practice to implement these methods (via your IDE)

EQUALS AND HASHCODE

- To generate equals and hashCode with IntelliJ follow these steps
  - Right-click your editor in the class for which you want to generate these methods
  - Select “Generate”
  - Select “equals() and hashCode()”
  - Select Next
  - Select all fields from the class for the equals()
  - Select Next
  - Select all fields from the class for the hasCode()
  - Select Create

EQUALS AND HASHCODE

- We already saw that "equals" and "hashCode" are important when using Set
- Other collections also use "equals" and "hashCode"
  - ArrayList in the method "contains"
  - HashMap in the method "get"
  - …
- It is best practice to always implement these methods for classes that you write
  - Use the default implementation from your IDE!
  - Without implementation, ”equals” will default to comparing memory references
- Otherwise, collections may exhibit unexpected behaviour

EXERCISE @CLASS

- exercise3
- exercise4



Collections

Optionals

Debugging

Optionals

OPTIONALS

```
public static void main(String[] args) {
    String word = findWordByDescription("a device used for calling people");

    System.out.println("The word is " + word);
    System.out.println("It's length is" + word.length());
}
```

- Java uses *null* if no value is present
- This can lead to unwanted errors if the developer forgets about the possibility of null
  - If the value of word is null, the code above will crash with a runtime error!

OPTIONALS

```
public static void main(String[] args) {
    Optional<String> word = findWordByDescription("a device used for calling people");

    if (word.isPresent()) {
        System.out.println("The word is " + word);
        System.out.println("It's length is" + word.get().length());
    }
}
```

- The built-in class Optional makes the possibility of null **explicit**
- It forces the developer to think about both cases:
  - What do I do if a value is present?
  - What do I do if a value is not present?

OPTIONALS – USAGE

Method	Description
Optional.empty()	**Static** method to create a new, empty optional
Optional.ofNullable(…)	**Static** method to wrap a value that could be null in an optional
isEmpty()/isPresent()	Check whether the optional contains a value
get()	Get the actual value from the optional, can only safely be used after checking presence first
orElseThrow(…)	Get the actual value from the optional, throw an exception if no value is present

- An optional is like a box that holds the actual value
  - As a developer you need to deal with the box to get to its contents
  - The value between <…> defines the type of the box its contents
    - Optional<User> holds a User class for example

Collections

Optionals

Debugging

Debugging

- Stack trace
- Debugging

STACK TRACE

```
Exception in thread "main" java.lang.IndexOutOfBoundsException: Index 20 out of bounds for length 4
 at java.base/jdk.internal.util.Preconditions.outOfBounds(Preconditions.java:64)
 at java.base/jdk.internal.util.Preconditions.outOfBoundsCheckIndex(Preconditions.java:70)
 at java.base/jdk.internal.util.Preconditions.checkIndex(Preconditions.java:266)
 at java.base/java.util.Objects.checkIndex(Objects.java:359)
 at java.base/java.util.ArrayList.get(ArrayList.java:427)
 at be.ucll.backend.users.repository.UserRepository.getOldestUser(UserService.java:43)
 at be.ucll.backend.users.MainMethod.main(MainMethod.java:28)
```

- If an exception is not caught in your own program and reaches the JVM your program crashes
- The JVM stops your program and prints the exception that led to the crash
  - This print happens in the form of a *stack trace*

STACK TRACE

```
Exception in thread "main" java.lang.IndexOutOfBoundsException: Index 20 out of bounds for length 4
 at java.base/jdk.internal.util.Preconditions.outOfBounds(Preconditions.java:64)
 at java.base/jdk.internal.util.Preconditions.outOfBoundsCheckIndex(Preconditions.java:70)
 at java.base/jdk.internal.util.Preconditions.checkIndex(Preconditions.java:266)
 at java.base/java.util.Objects.checkIndex(Objects.java:359)
 at java.base/java.util.ArrayList.get(ArrayList.java:427)
 at be.ucll.backend.users.repository.UserRepository.getOldestUser(UserService.java:43)
 at be.ucll.backend.users.MainMethod.main(MainMethod.java:28)
```

- Reading the stack trace from top to bottom shows all the methods the exception passed through on its way out of your program
  - Not every method encountered will be code that you wrote
  - The end of the stack trace will always be main, the final method in our program before reaching the JVM
- Looking for your own code in a stack trace is good way to find parts of your program that are generating unwanted errors (called *bugs*)

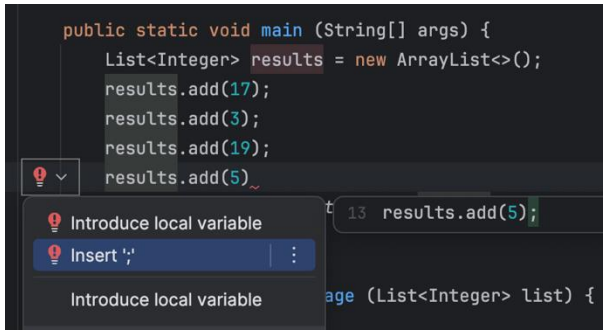
BUG

"First actual case of bug being found."

The story goes that on September 9th, 1947, computer scientist Grace Hopper found a moth in the Harvard Mark II computer's log book and reported the world's first literal computer bug. However, the term "bug", in the sense of technical error, dates back at least to 1878 and with Thomas Edison.

DEBUGGING

- Is the process of detecting and correcting errors in a program
- There are different kinds of errors:
  - **Compilation errors**: errors that are caught by the compiler before running your program, no debugging necessary!



  - **Runtime errors**: errors that only occur when your program is running, sometimes they are easily debugged by looking at the stack trace

```
/Users/elke/Library/Java/JavaVirtualMachines/corretto-17.0.10/Contents/Home/bin/java -javaagent:/Applications/IntelliJ IDEA.app/Cont
Exception in thread "main" java.lang.NullPointerException Create breakpoint : Cannot invoke "String.length()" because "word" is null
    at Optionals.OptionalNullPointer.main(OptionalNullPointer.java:9)

Process finished with exit code 1
```

  - **Logical runtime errors**: some runtime errors can take long to find; this is where debugging is useful …

DEBUGGER

- Is a tool that lets you find bugs in an efficient manner by providing an insight into the internal operations of a program
  - This is possible by pausing the execution at a specified point, analysing the program state, and, if necessary, advancing the execution step-by-step

EXAMPLE

```java
public class Average {

    public static void main (String[] args) {
        List<Integer> results = new ArrayList<>();
        results.add(17);
        results.add(3);
        results.add(19);
        results.add(5);
        System.out.println(getAverage(results));
    }

    public static int getAverage (List<Integer> list) {...}

}
```

```
/Users/elke/Library/Java/JavaVirtualMa
1

Process finished with exit code 0
```

BREAKPOINT

- You set a breakpoint at the line of code where you think the bug is situated
- The breakpoint indicates where the program will be suspended for you to examine its state

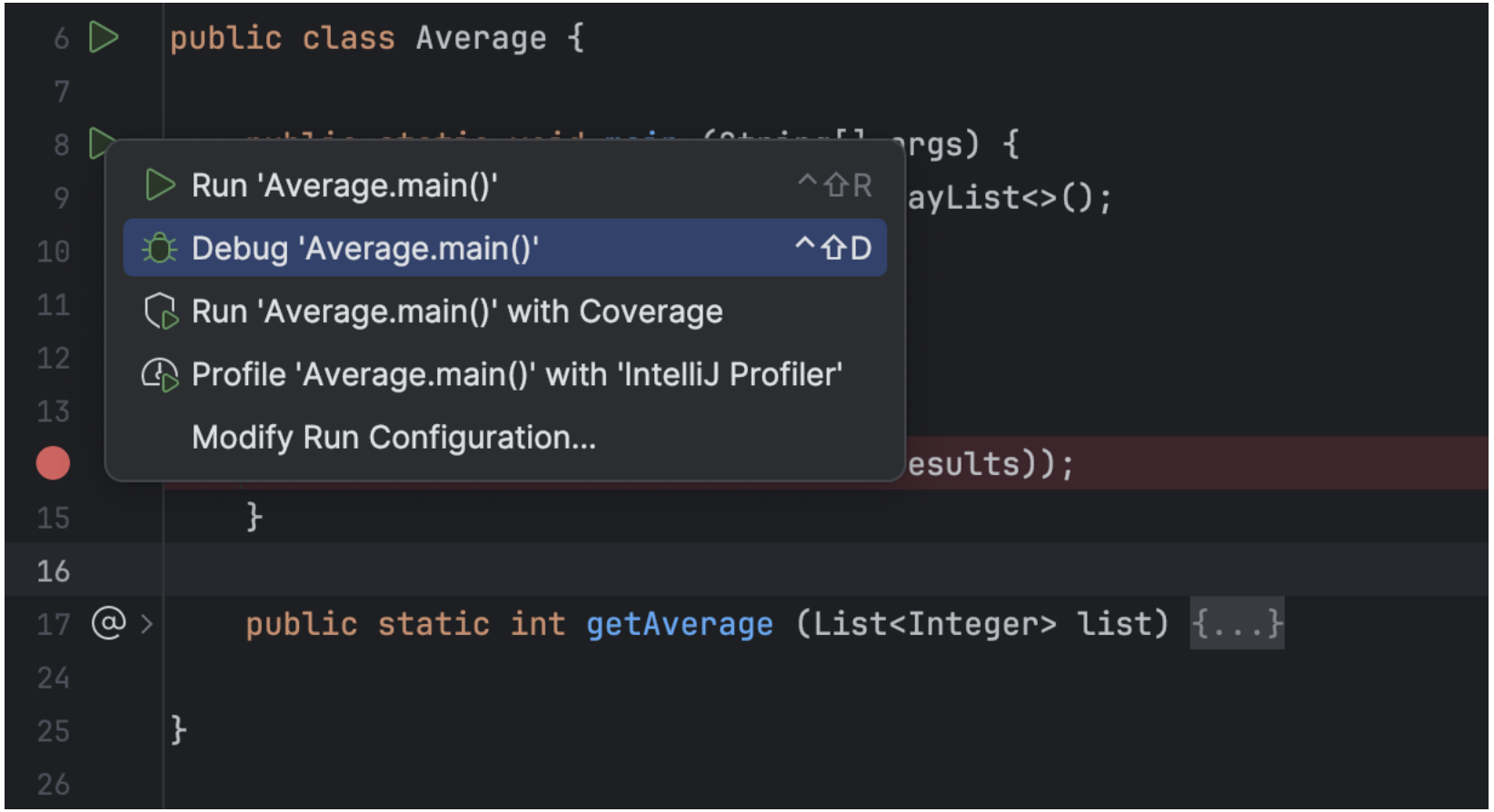
EXAMPLE – SET A BREAKPUNT

```
6  public class Average {
7
8      public static void main (String[] args) {
9          List<Integer> results = new ArrayList<>();
10         results.add(17);
11         results.add(3);
12         results.add(19);
13         results.add(5);
           System.out.println(getAverage(results));
15     }
16
17     public static int getAverage (List<Integer> list) {...}
24
25 }
```

START DEBUG MODE

- You need to start the program in debug mode

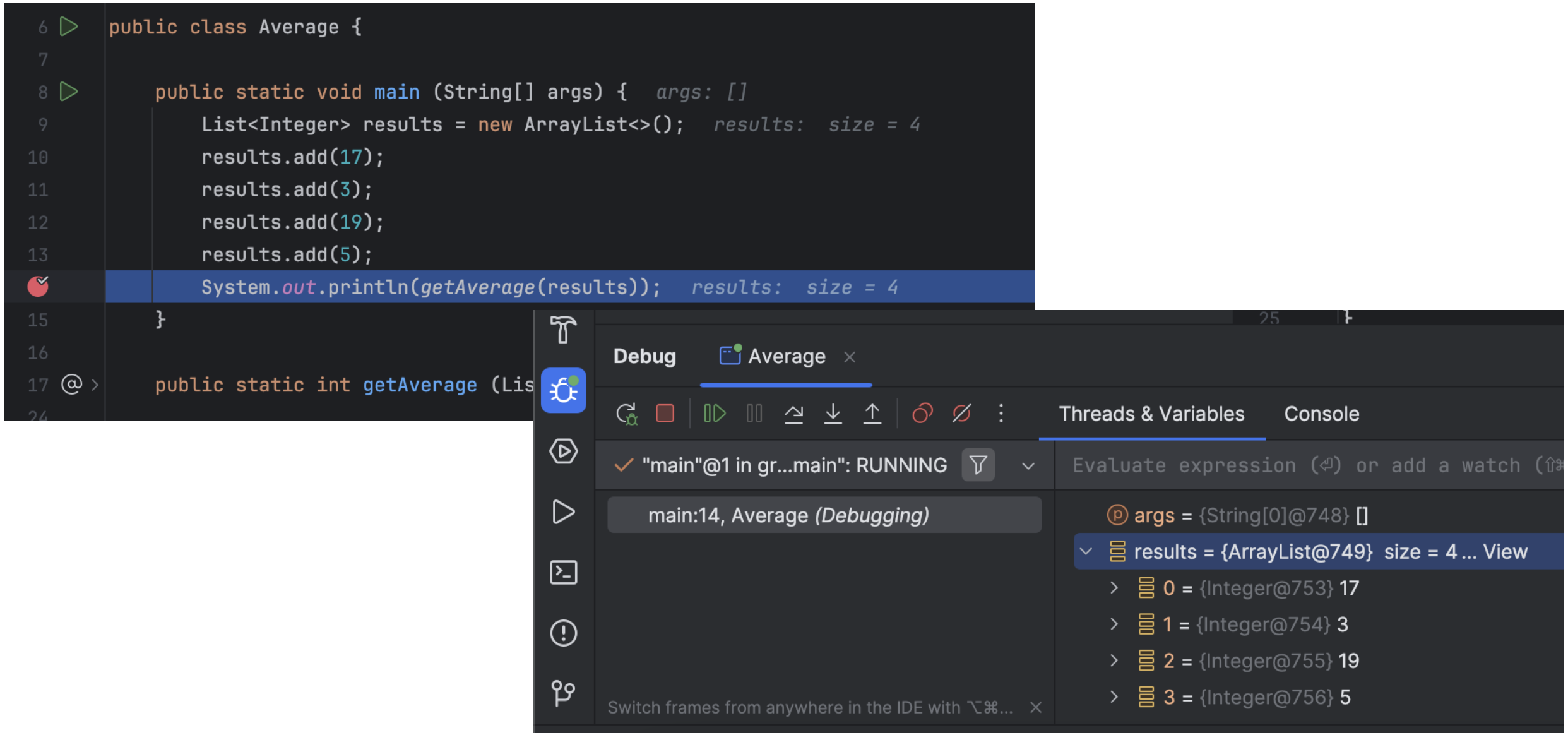
EXAMPLE – START DEBUG MODE



INSPECT

- After the debugger session has started, the program runs normally until a breakpoint is hit
  - When this happens, IntelliJ IDEA pauses the program, highlights the line, at which the program is suspended, and shows the Debug tool window

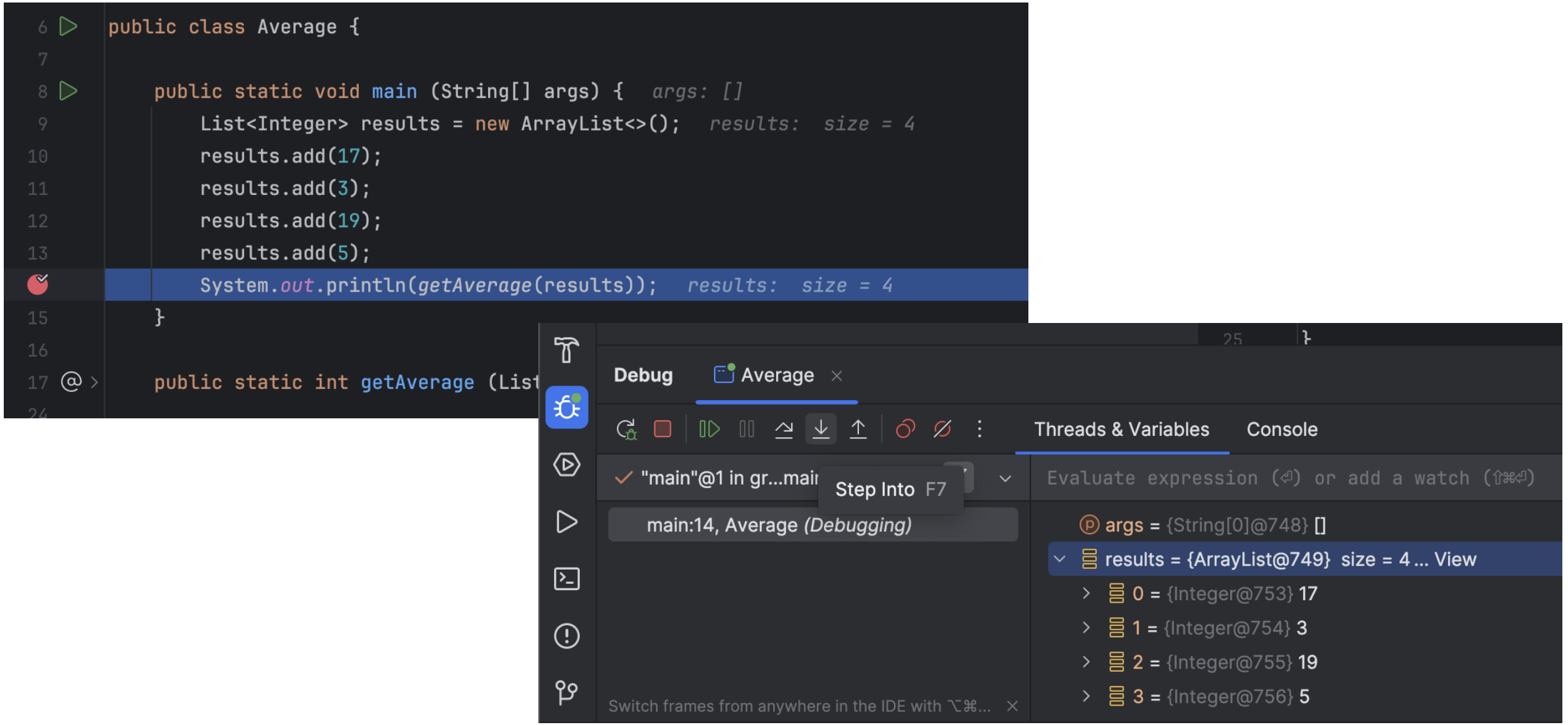
EXAMPLE - INSPECT THE VARIABLES



STEP INTO

- Enables you to step into a method, an iteration, … to inspect in detail what happens in this part of the code

EXAMPLE – STEP INTO (METHOD)

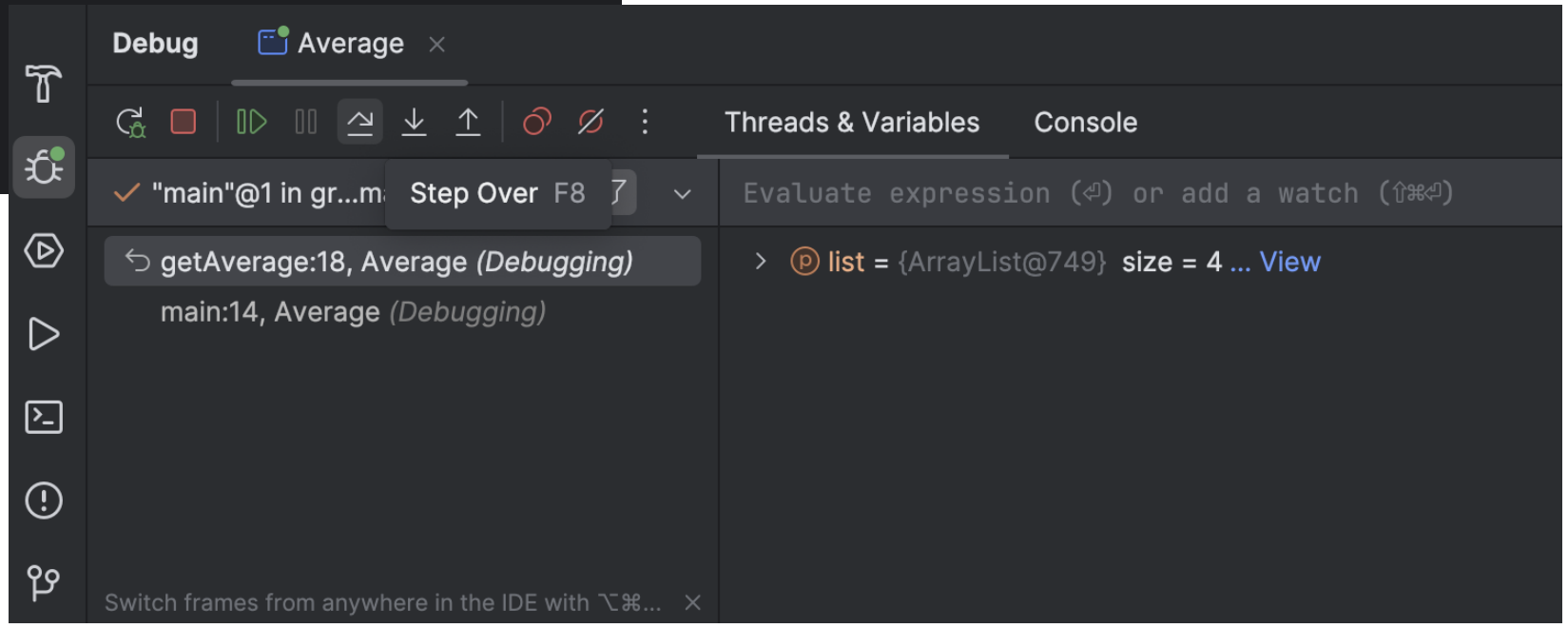


STEP OVER

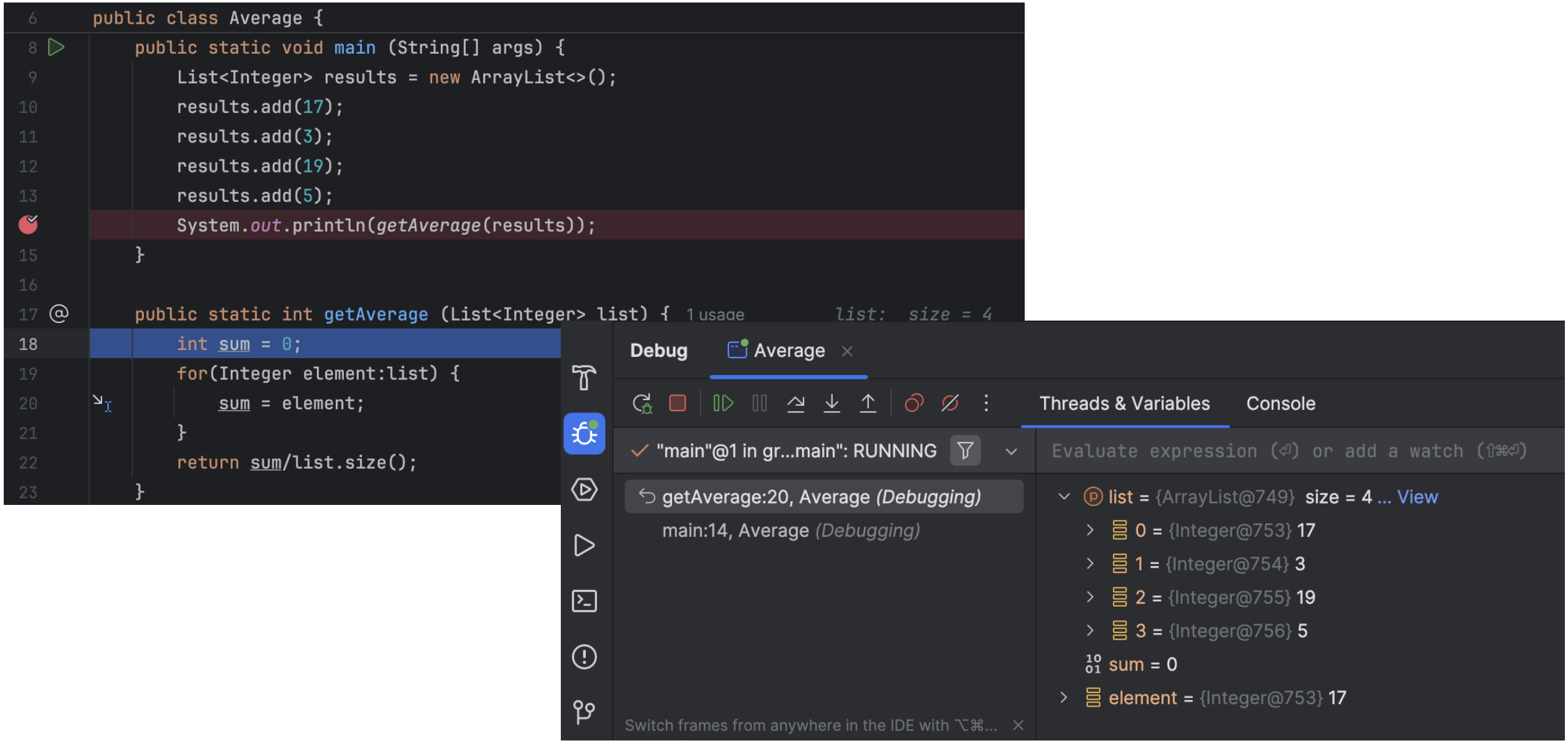
- Enables you to decide to not step into a method, an iteration, … because you don’t think the bug is in this piece of code

EXAMPLE – STEP OVER

```
6   public class Average {
8       public static void main (String[] args) {
9           List<Integer> results = new ArrayList<>();
10          results.add(17);
11          results.add(3);
12          results.add(19);
13          results.add(5);
            System.out.println(getAverage(results));
15      }
16
17      public static int getAverage (List<Integer> list) {  1 usage        list:  size = 4
18          int sum = 0;
19          for(Integer element:list) {
20              sum = element;
21          }
22          return sum/list.size();
23      }
```



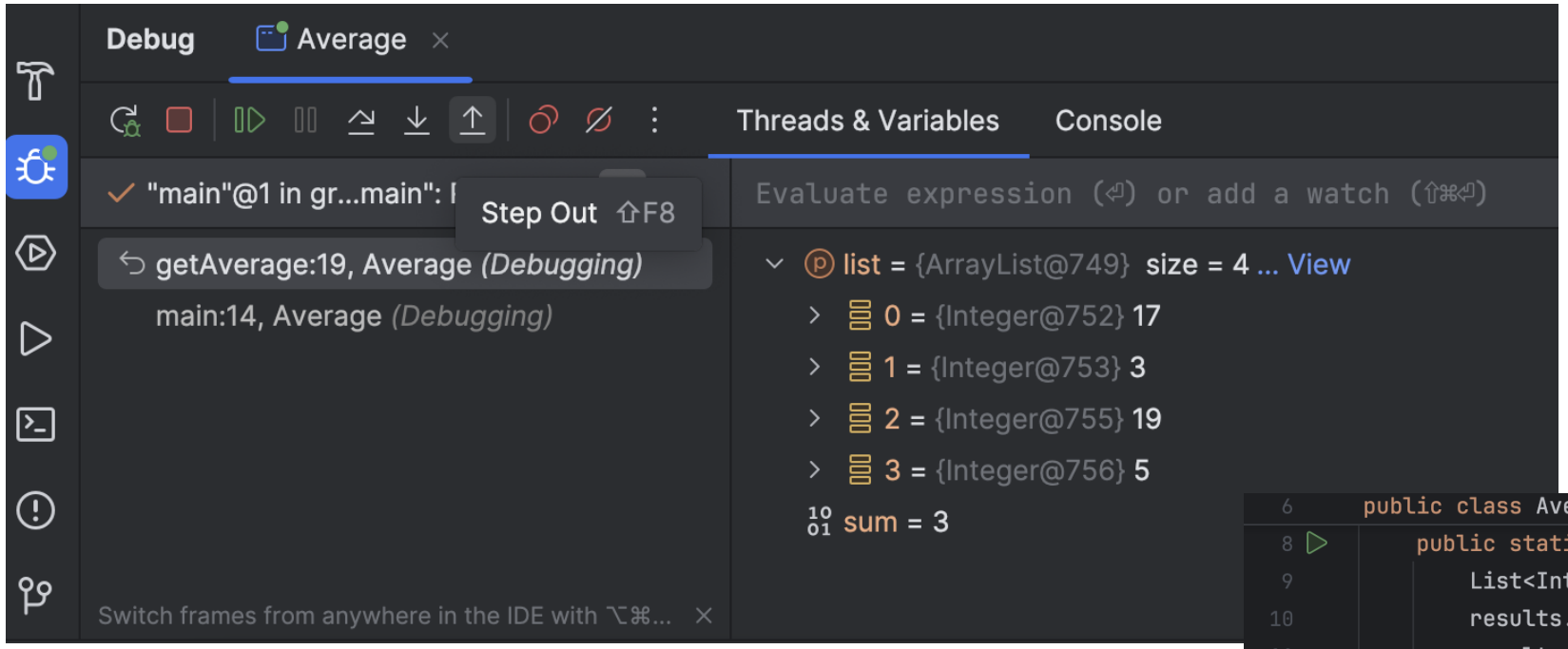
EXAMPLE – STEP INTO (FOR)



STEP OUT

- Enables you to decide to step out of a method, an interation, … which you stepped into because you found the bug or this piece of code is correctly behaving and thus the bug isn’t in this piece of code

EXAMPLE – STEP OUT




```
6   public class Average {
8       public static void main (String[] args) {  args: []
9           List<Integer> results = new ArrayList<>();  results:  size = 4
10          results.add(17);
11          results.add(3);
12          results.add(19);
13          results.add(5);
            System.out.println(getAverage(results));  results:  size = 4
15      }
16
17      public static int getAverage (List<Integer> list) {  1 usage
18          int sum = 0;
19          for(Integer element:list) {
20              sum = element;
21          }
22          return sum/list.size();
23      }
```

DEBUGGING IN INTELLIJ

- For more information see
  - https://www.jetbrains.com/help/idea/debugging-your-first-java-application.html

EXERCISE @CLASS

- exercise9
- exercise10
- exercise11
- exercise12



EXERCISE @HOME

- exercise1
- exercise2
- exercise5
- exercise6
- exercise7
- exercise8
- exercise13



