Programming II

Dominique Brodbeck & Rahel Lüthy · Fachhochschule Nordwestschweiz · 2016

Version 1.1.1

Table of Contents

Preamble

Motivation

This document gives a very dense summary of what we consider the most important topics of the "Programming II" course. It is meant as a starting point, explaining just the bare minimum. **You will have to do some further reading** – tutorials, books, articles, API documentation.

Everyone is different! We don't want to force you to read a specific book. That's why we leave it up to you to find documentation that suits your preferred style. Some people love reading a language specification, others learn best from examples. Some students read books cover to cover, while others hop from one Stack Overflow answer to the next.

Each chapter ends with a "Resources" section which gives helpful links. We don't specifically recommend an official course book, but the following three might be of interest:

Sprechen Sie Java? by Hanspeter Mössenböck (5. Auflage) – The official "Programming I" course book has a certain overlap with "Programming II". We will link to the relevant sections wherever applicable.

Java Programming for Kids by Yakov Fain (free online version) – Once you get over the humiliation of reading a book for children, you will appreciate its simple and clear style.

Effective Java by Josh Bloch – A very advanced book which does not focus on Java basics, but on writing clear, correct, robust, and reusable code.

Authors

Rahel Lüthy

Rahel Lüthy is a software developer and research associate at the Institute for Medical and Analytical Technologies (FHNW). She has an MSc in evolutionary & population biology (University of Basel), and 15+ years of industry experience as a Java developer. Apart from Java, she is passionate about Scala, functional programming, open-source, Git, and Gradle. A dog, two kids, three karate belts, four bicycles, and five guitars keep her balanced.

Dominique Brodbeck

Dominique Brodbeck is a professor for biomedical informatics at the Institute for Medical and Analytical Technologies (FHNW), as well as founding partner of the company Macrofocus GmbH. He has a PhD in Physics (University of Basel) and holds an Executive MBA in Management of Technology (EPFL/HEC Lausanne). His activities focus on how to extract meaningful information from large amounts of complex data, and on how to make it accessible to humans in a usable way. His motto is: "I want to make people happy by making computers sing and dance". He owns seven bicycles, following rule #12: The correct number of bikes to own is n+1, where n is the number of bikes currently owned.

Chapter 1. Introduction

In this chapter you will learn how to set up your development environment (IDE) in order to work on the course exercises.

1.1. Test-Driven Development

Throughout this course, you will be asked to solve exercises for each chapter. The organization of exercises is inspired by an idea called *Test-Driven Development*: You receive automated JUnit tests that define the desired outcome of your exercises, and are asked to implement a solution that passes these tests.

Initially, all tests are failing ("red"). You know that you are done with your exercises as soon as all tests are succeeding ("green").

For each course chapter, you receive a ZIP containing a Java project with the following main elements:

- A Gradle configuration which allows to conveniently import the project into any IDE (e.g. IntelliJ IDEA)
- A `src/main/java` folder which contains empty stubs for your solutions
- A `src/main/test` folder which contains JUnit tests

1.2. Exemplary Walk-Through

This introductory chapter will walk you through the steps needed to set up an exemplary project. At the end, you will be all set for working on the "real" exercises later.

1.2.1. Download & IDE Import

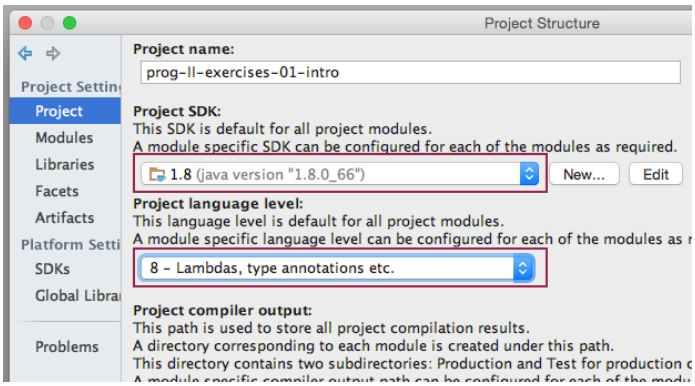
Download the prog-II-exercise-01-intro.zip and extract it to any folder. The directory tree contains a variety of files, but first, we are only interested in the `build.gradle` file. This is a configuration file for Gradle, a build tool that is supported by all major IDEs (IntelliJ IDEA, Eclipse, NetBeans). A build file describes the structure of the project. It declares which programming language is being used (`java`) and specifies where the actual code is located (e.g. the `src/main/java` folder). You do **not** need to know Gradle in order to successfully complete this course. Gradle is only used to allow for a smooth import of exercise projects into your IDE.

This guide assumes that you have a working installation of JDK 8 and IntelliJ IDEA 15. Most setup steps are similar with Eclipse or NetBeans – feel free to use your IDE of choice.

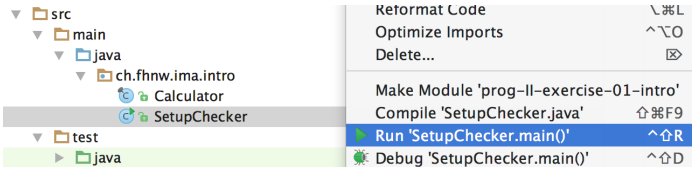
To import the project into IntelliJ IDEA, invoke the `File > New > Project From Existing Sources` action and select the `build.gradle` file.

When starting with a very fresh IntelliJ IDEA download, you might run into `JAVA_HOME` issues. This Stack Overflow article explains how to resolve them. From the initial wizard, proceed with `Open⋯` to select the `build.gradle` file and import the project.

Trust the default settings and press OK to trigger the import. Once imported, make sure that IntelliJ IDEA uses JDK 8 to compile your Java code. Open `File > Project Structure` to display the settings:



Finally, you can verify that your setup is correct by compiling the project (`Build > Make Project`) and running the `SetupChecker` class:



If the text "Hello World" appears in your `Run` console window, everything is set up correctly – congratulations!

You don't need to understand the `SetupChecker` code. This class just verifies that your IDE is correctly configured to compile & run Java 8 projects.

1.2.2. JUnit

In this introductory example, you are asked to program a simple calculator. The `CalculatorTest` class contains JUnit tests that serve as a specification, i.e. they declare how your `Calculator` implementation shall behave.

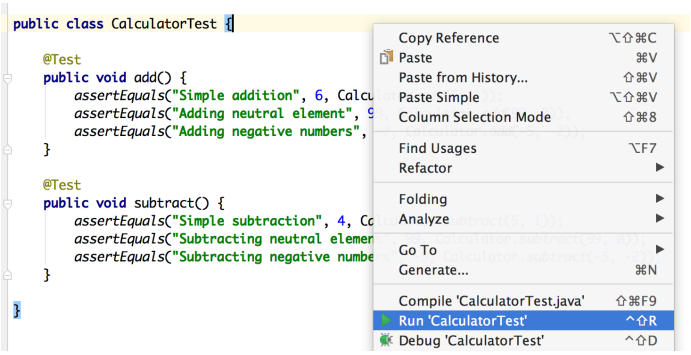
JUnit is a wide-spread framework to write unit tests in Java. Generally speaking, unit tests assert that a unit of code behaves according to expectations (in Java, a unit corresponds to a class). Unit testing has a lot of benefits and learning to use JUnit is well worth the effort.

However, in this course you will not actually need to write JUnit tests. The tests are provided to guide you towards a correct implementation of each exercise, i.e. they drive your implementation, hence the term *Test-Driven Development*.

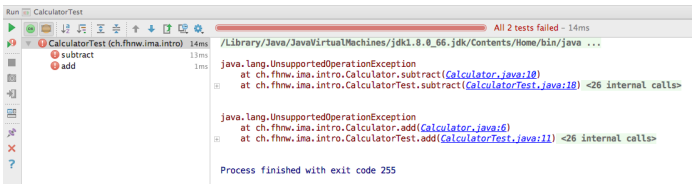
While the `CalculatorTest` class contains a full suite of tests, the `Calculator` class itself does not contain the correct implementation yet, it only contains method stubs.

It is a convention to name the test class after the class which is being tested: The class `CalculatorTest` contains all tests for the class `Calculator`. To keep the main code (`Calculator`) well separated from the test code (`CalculatorTest`), the top-level `src` folder is split into two distinct folder hierarchies (`test` vs `main`).

The `CalculatorTest` can be run via context menu:



This will execute all tests in a JUnit-specific runner. As expected, they all fail because the `Calculator` is not yet implemented:



1.2.3. Code!

Now is the time to actually work on the exercises. Roll back your sleeves and implement the `Calculator` class (do **not** change the `CalculatorTest` class).

Inside `Calculator`, delete all lines which look like this

```
throw new UnsupportedOperationException();
```

and replace them with the correct calculation logic. Verify the correctness of your implementation by re-running the JUnit tests. You are done once all tests are green ☺

1.3. Resources

JDK 8

http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html

IntelliJ IDEA

https://www.jetbrains.com/idea

JUnit

http://junit.org

Gradle

http://gradle.org

Chapter 2. Data Structures

In "Programming I" you learned how to use primitive data types like `boolean`, `int`, or `float`. You also learned how to store multiple values in the form of arrays, e.g. `boolean[]`, `int[]`, or `float[]`. An array represents a very basic data structure, i.e. a particular way of organizing data in a program.

In this chapter you will learn about other data structures, namely `List`, `Set`, and `Map`, which offer certain advantages over using basic arrays. Choosing the right data structure is crucial to making programs efficient in terms of execution speed and memory usage.

2.1. The Array Data Structure

Data structures group multiple objects into a single unit. In Java, the most basic container that can hold a fixed number of values is an array:

```
// creates an array of strings and initializes it with three values
String[] fruit = {"Apple", "Orange", "Banana"};

String someFruit = fruit[1];   // index-based access
System.out.println(someFruit); // prints "Orange"
```

2.2. Array Alternatives

It seems that arrays are simple to use and need very little memory – so why ever use a different data structure?

Imagine that your program needs to read values from a file. When you write the program, you do not know how many values the file will contain, so how big should you initialize your array?

> **Solution:** The `List` data structure provides an array-alternative which grows dynamically whenever new elements are added.

Imagine that your program needs to read some text and output all unique characters used in that text. When using an array to store the characters, how would you make sure that you do not store the same character twice? Checking the complete array every time you encounter a character will make your program very slow. And how big should you initialize your array?

> **Solution:** The `Set` data structure provides an array-alternative that grows dynamically when new elements are added and makes sure that it never contains duplicates.

Imagine that your program needs to read some text and create a histogram for all characters used in

that text, i.e. it should gather statistics in the form of `'a: 33', 'b: 0', 'c: 12'` etc. Of course you can use two arrays, one to keep track of the characters, and one to keep track of the counts. But how big should you initialize them? If someone was asking just for the count of the `n` characters, how would you look that up efficiently?

> **Solution:** The `Map` data structure provides an array-alternative that maps keys to values (in our example, you would use characters as keys and counts as values). A set grows dynamically whenever a new mapping is added and provides very fast look-up operations.

The following sections will explain `List`, `Set`, and `Map` in more detail.

2.3. The List Interface

The `List` interface provides a sequential data structure which grows dynamically whenever new elements are added.

Don't worry if you are not familiar with the term "interface" – you will learn more about this concept in a future chapter. For the time being, think of interfaces as a simple contract: They specify a group of methods that an implementation of that interface must provide.

The `ArrayList` class represents a general-purpose implementation of the `List` interface. Here's how an `ArrayList` can be created, filled, and queried:

```
List<String> fruit = new ArrayList<>();

System.out.println(fruit.size()); // prints 0

fruit.add("Apple");
fruit.add("Orange");
fruit.add("Banana");

System.out.println(fruit.size()); // prints 3
String someFruit = fruit.get(1);  // index-based access
System.out.println(someFruit);    // prints "Orange"
```

The angle brackets used in the `List<String>` declaration can be read as *of element type String*. The concept behind this syntax is called "generics" or "type parameterization". In our example, it guarantees that the `add` and `get` methods only allow elements of type `String`. Because we declare our `fruit` variable to be of type `List<String>` the compiler can tell that our `ArrayList` must also be of element type String, therefore we don't have to repeat the element type but can use empty brackets `<>` ("diamond") instead.

As mentioned above, `List` is an interface (a contract), while `ArrayList` is a concrete implementation (a class that fulfills the contract by supporting all required methods). It is good practice to program to the interface rather than the implementation:

```
List<String> list = new ArrayList<>();             // good
ArrayList<String> anotherList = new ArrayList<>(); // bad
```

This assures that wherever you use your `list` variable, you are only using methods that are actually declared by the `List` interface. If you later on decide to e.g. use a `LinkedList` (an alternative implementation of the `List` interface) rather than an `ArrayList`, you only need to swap out the implementation in one spot.

The easiest way to iterate over all elements of a list is by using a so-called "for-each" loop:

```
List<String> list = Arrays.asList("Apple", "Orange", "Banana");

for (String element : list) {
    System.out.println(element);
}
```

Using `Arrays.asList("Apple", "Orange", "Banana")` is a simple way to create a fixed-size list with some elements

Alternatively, all lists can also be converted to data streams, which can then be processed via The Java 8 Stream API.

You are now ready to tackle the first part of the exercises. Get yourself a coffee, import the `build.gradle` file contained inside prog-II-exercise-02-data-structures.zip, and get started with implementing the `List101` class.

2.4. The Set Interface

The `Set` interface provides a data structure that grows dynamically when new elements are added and makes sure that it never contains duplicates.

The `HashSet` class is the most commonly used `Set` implementation. The following example illustrates how a `HashSet` can be used:

```
Set<String> fruit = new HashSet<>();

System.out.println(fruit.size());           // prints 0

fruit.add("Apple");
fruit.add("Orange");
fruit.add("Banana");

System.out.println(fruit.size());           // prints 3

fruit.add("Orange");

System.out.println(fruit.size());           // still prints 3

System.out.println(fruit.contains("Apple")); // prints true
System.out.println(fruit.contains("Lemon")); // prints false
```

As mentioned before, the angle brackets in the `Set<String>` declaration can be read as *of element type String*

While we can also use a "for-each" loop to iterate over all elements, the `HashSet` class makes no guarantees as to the **order** of the iteration:

```
Set<String> words = new HashSet<>(Arrays.asList("Apple", "Banana", "Orange"));
for (String word : words) {
    System.out.println(word);
}
```

This will print elements in unpredictable order:

```
Apple
Orange
Banana
```

The `LinkedHashSet` class represents a `Set` implementation which guarantees that elements are returned in the order in which they were inserted. However, this convenience comes at a certain price, i.e. a `LinkedHashSet` uses more memory than a traditional `HashSet`.

2.5. The Map Interface

The `Map` interface provides a data structure which maps keys to values. A map will never contain duplicate keys (i.e. the keys behave like a set), and each key can map to at most one value.

The `HashMap` class is the most commonly used `Map` implementation. The following example illustrates how a `HashMap` can be used:

```
Map<String, Integer> birthYears = new HashMap<>();

System.out.println(birthYears.size());                // prints 0

birthYears.put("Barack", 1961);
birthYears.put("Hillary", 2016);
birthYears.put("Ada", 1815);

System.out.println(birthYears.size());                // prints 3

birthYears.put("Hillary", 1947);

System.out.println(birthYears.size());                // still prints 3

System.out.println(birthYears.containsKey("Hillary")); // prints true
System.out.println(birthYears.get("Hillary"));         // prints 1947
System.out.println(birthYears.containsKey("Donald"));  // prints false
```

As mentioned before, the angle brackets in the `Map<String, Integer>` declaration specify the type of entries added to the map. For maps, we need two type declarations, one for the keys and one for the values. In this example, the key are of type `String` while the values are of type `Integer`, but you might as well construct a map where keys and values are of the same type (`Map<String, String>`).

Each map offers three different views on its contents: it can be queried for a set of keys, a collection of all its values, or a set of its entries:

```
Map<String, Integer> birthYears = new HashMap<>();
birthYears.put("Barack", 1961);
birthYears.put("Hillary", 1947);
birthYears.put("Ada", 1815);

Set<String> keys = birthYears.keySet();
System.out.println(keys);    // prints [Hillary, Barack, Ada]

Collection<Integer> values = birthYears.values();
System.out.println(values);  // prints [1947, 1961, 1815]

Set<Map.Entry<String, Integer>> entries = birthYears.entrySet();
System.out.println(entries); // prints [Hillary=1947, Barack=1961, Ada=1815]
```

Iteration order is not guaranteed. A `LinkedHashMap` can be used to return contents in key-insertion order.

The `entrySet()` view is particularly helpful when iterating over the contents of a map:

```
Map<String, Integer> birthYears = new HashMap<>();
birthYears.put("Barack", 1961);
birthYears.put("Hillary", 1947);
birthYears.put("Ada", 1815);

for (Map.Entry<String, Integer> entry : birthYears.entrySet()) {
    System.out.println(entry.getKey() + " -> " + entry.getValue());
}
```

This will print the following lines (in unpredictable order):

```
Hillary -> 1947
Barack -> 1961
Ada -> 1815
```

If a `Map` is queried for a key it does not contain, a `null` value will be returned. Working with `null` values is very error prone and should be avoided whenever possible. Java 8 offers a safe way of providing a default value for absent mappings:

```
Map<Integer, String> numbers = new HashMap<>();
numbers.put(0, "zero");
numbers.put(2, "two");
numbers.put(3, "three");

String defaultValue = "unknown";

System.out.println(numbers.getOrDefault(0, defaultValue)); // prints "zero"
System.out.println(numbers.getOrDefault(1, defaultValue)); // prints "unknown"
```

2.6. The Collections Helper

The `List`, `Set`, and `Map` interfaces all belong to the Java Collections Framework, which provides a variety of data structures ready to be used.

In addition, the framework also offers the `Collections` helper class. Among its many utility methods, `min` and `max` are particularly helpful when working with numbers:

```
List<Integer> values = Arrays.asList(1, 99, -2, 42);
System.out.println(Collections.min(values)); // prints -2
System.out.println(Collections.max(values)); // prints 99
```

2.7. The Java 8 Stream API

The previous sections explained how `List`, `Set`, and `Map` data structures all belong to a common hierarchy of collections. Some collections allow duplicate elements and others do not. Some are ordered and others unordered. But generally speaking, a collection stores a number of elements in memory.

Java 8 introduced a further abstraction, namely the `Stream` interface – a data source which sequentially serves elements without storing them in memory. The concept of streams is very powerful, but also quite complex. The vast number of available online tutorials is a clear indication of this fact. We will not cover streams in detail in this course, but this introductory section gives a little taste of their power.

In contrast to collections, streams do not store elements in memory but simply serve as a source of elements. Some streams provide elements by reading them from a collection, but others read them from the internet, or calculate them on the fly. Much like a water pipeline, where water only flows if the faucet is turned open, streams only serve elements as long as these elements are actually consumed.

At least three aspects of streams deserve special attention:

- Streams offer a variety of methods to filter, transform, and collect elements. These methods can be chained in a very functional way, which leads to concise, but readable code.
- Streams can be created **from** collections and converted back **to** collections.
- Streams serve as a potentially infinite source of elements.

Let's dive in with an example. Given a list of words, produce a list of `Integer` values which correspond to each word's length:

```
Stream<String> words = Stream.of("hello", "new", "world", "of", "streams");
List<Integer> lengths = words.map(word -> word.length()).collect(Collectors.toList());
System.out.println(lengths); // prints [5, 3, 5, 2, 7]
```

In order to understand a sequence of instructions, it is often helpful to look at the types of intermediate results:

```
Stream<String> words = Stream.of("hello", "new", "world", "of", "streams");
List<Integer> lengths = words.        // Starting with a Stream<String> ...
        map(word -> word.length()).   // ... transform each String to an Integer
        collect(Collectors.toList()); // ... convert the final Stream<Integer> to a
List<Integer>
```

Just for illustration purposes – calculating the maximum length of a filtered list of words:

```
Stream<String> words = Stream.of("Apple", "Orange", "Apricot", "Banana", "Orange");
int maxLength = words.
        filter(word -> word.startsWith("A")). // [Apple, Apricot]
        mapToInt(word -> word.length()).      // [5, 7]
        max().getAsInt();                     // 7
System.out.println(maxLength);                // prints 7
```

All collections offer a `stream()` method which returns a sequential stream over their elements. The following example shows how to create a stream **from** a `List`, process its elements, and convert it **to** a `Set`:

```
List<String> words = Arrays.asList("Apple", "Orange", "Apricot", "Banana", "Orange");
Set<String> filteredWords = words.stream().
        filter(word -> word.contains("r")).
        collect(Collectors.toSet());
System.out.println(filteredWords); // prints [Apricot, Orange]
```

The `Collectors.toSet()` method internally creates a `HashSet` to store the final elements. That's why the final `Set` does not contain elements in order. This can however be achieved by explicitly collecting elements inside a `LinkedHashSet`:

```
List<String> words = Arrays.asList("Apple", "Orange", "Apricot", "Banana", "Orange");
Set<String> filteredWords = words.stream().
        filter(word -> word.contains("r")).
        collect(Collectors.toCollection(LinkedHashSet::new));
System.out.println(filteredWords); // prints [Orange, Apricot]
```

To round off this introductory stream section, the following example uses an infinite stream to create random dice rolls:

```
Random random = new Random();
Stream<Integer> diceRollingStream = Stream.generate(() -> random.nextInt(6) + 1);
List<Integer> threeDiceRolls = diceRollingStream.limit(3).collect(Collectors.toList());
```

2.8. Resources

Hanspeter Mössenböck: Sprechen Sie Java?

Kapitel 15: "Generizität", Kapitel 23.1: "Collection-Typen"

The Java Tutorials: Collections

https://docs.oracle.com/javase/tutorial/collections/

Angelika Langer: Java Generics FAQs

http://www.angelikalanger.com/GenericsFAQ/FAQSections/ProgrammingIdioms.html

Benjamin Winterberg: Java 8 Stream Tutorial

http://winterbe.com/posts/2014/07/31/java8-stream-tutorial-examples/

Oracle Technology Network: Processing Data with Java SE 8 Streams

http://www.oracle.com/technetwork/articles/java/ma14-java-se-8-streams-2177646.html

Chapter 3. Input / Output

In this chapter you will learn how to read and write textual data. You will use the `Scanner` class to read textual input, and the `PrintWriter` class to output data back to file.

3.1. Reading Text with a Scanner

The `Scanner` class allows to read textual input:

```
String input = "Mary had a little lamb";

Scanner scanner = new Scanner(new StringReader(input));
while (scanner.hasNext()) {
    System.out.println(scanner.next());
}
```

This will print:

```
Mary
had
a
little
lamb
```

Each `Scanner` instance can be configured in how it breaks input into tokens. Various `hasNextXXX()` and `nextXXX()` methods allow to parse different value types. The following example demonstrates how to parse `double` values separated by a dash (`-`):

```
String input = "0.1-42.0-99.9";

Scanner scanner = new Scanner(new StringReader(input));
scanner.useDelimiter("-");
List<Double> values = new ArrayList<>();
while (scanner.hasNextDouble()) {
    values.add(scanner.nextDouble());
}
System.out.println(values);
```

Multi-line input can be parsed by making use of the `hasNextLine()` and `nextLine()` methods:

```
String input =
        "line 1\n" +
        "line 2\n" +
        "line 3";

Scanner scanner = new Scanner(new StringReader(input));
while (scanner.hasNextLine()) {
    System.out.println(scanner.nextLine());
    System.out.println("------");
}
```

Which will print:

```
line 1
------
line 2
------
line 3
------
```

In all examples so far we were reading input from a `String` variable. In reality, data is often read from a file on disk. The `File` class can be used to represent such a file. Note that the actual scanning logic stays exactly the same as before. However, things can go wrong when working with files, that's why a `try/catch` exception handling construct is needed:

```
File inputFile = new File("input.txt");
try (Scanner scanner = new Scanner(inputFile)) {
    while (scanner.hasNext()) {
        System.out.println(scanner.next());
    }
} catch (FileNotFoundException e) {
    System.err.println("Reading file failed: " + e.getMessage());
}
```

Have a look at the official Oracle Tutorial if you would like to learn more about the concept of exceptions.

3.2. Creating & Formatting Text

When dealing with text, `String` values are often created via successive concatenation, i.e. by repeatedly appending fragments. This can most elegantly be achieved via the `StringBuilder` class:

```
StringBuilder result = new StringBuilder();
for (String name : Arrays.asList("Lisa", "Bob", "Hillary")) {
    result.append(name).append("\n");
}
System.out.println(result);
```

Which will print:

```
Lisa
Bob
Hillary
```

The `String.format` helper method allows to write out values in a specific format:

```
List<String> numbers = Arrays.asList("one", "two", "three");
StringBuilder result = new StringBuilder();
for (int i = 0; i < numbers.size(); i++) {
    String formatted = String.format("%d %S", i, numbers.get(i));
    result.append(formatted).append("\n");
}
System.out.println(result);
```

Which will print:

```
0 ONE
1 TWO
2 THREE
```

3.3. Writing Text with a PrintWriter

The `PrintWriter` class can be used to write text to a file:

```
File outputFile = new File("out.txt");
try (PrintWriter writer = new PrintWriter(outputFile)) {
    writer.print("Hello World");
} catch (FileNotFoundException e) {
    System.err.println("Writing file failed: " + e.getMessage());
}
```

As seen before, interacting with the `File` class can lead to exceptions. The `try/catch` clause is taking care of properly handling the error scenario. In addition, constructing the `PrintWriter` in the `try()`

statement makes sure that it is properly flushed and closed once we are done working with it.

3.4. Resources

Scanner

http://docs.oracle.com/javase/8/docs/api/java/util/Scanner.html

File

http://docs.oracle.com/javase/8/docs/api/java/io/File.html

Exceptions

https://docs.oracle.com/javase/tutorial/essential/exceptions/index.html

StringBuilder

http://docs.oracle.com/javase/8/docs/api/java/lang/StringBuilder.html

Format String Syntax

http://docs.oracle.com/javase/8/docs/api/java/util/Formatter.html#syntax

PrintWriter

http://docs.oracle.com/javase/8/docs/api/java/io/PrintWriter.html

try-with-resource

https://docs.oracle.com/javase/tutorial/essential/exceptions/tryResourceClose.html