Programming II

Dominique Brodbeck & Rahel Lüthy · Fachhochschule Nordwestschweiz · 2016

Version 1.4.0

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

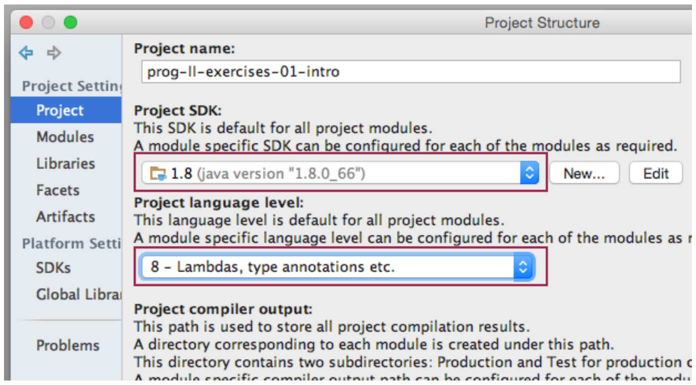
Download the prog-II-exercise-01-intro.zip and extract it to any folder. The directory tree contains a variety of files, but first, we are only interested in the `build.gradle` file. This is a configuration file for Gradle, a build tool that is supported by all major IDEs (IntelliJ IDEA, Eclipse, NetBeans). 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:



Project Structure

Project Settin
Project
Modules
Libraries
Facets
Artifacts
Platform Setti
SDKs
Global Librai
Problems

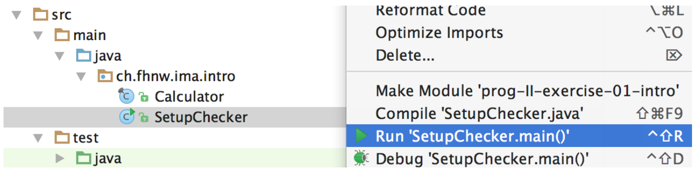
Project name:
prog-II-exercises-01-intro

Project SDK:
This SDK is default for all project modules.
A module specific SDK can be configured for each of the modules as required.
1.8 (java version "1.8.0_66")   New...   Edit

Project language level:
This language level is default for all project modules.
A module specific language level can be configured for each of the modules as r
8 - Lambdas, type annotations etc.

Project compiler output:
This path is used to store all project compilation results.
A directory corresponding to each module is created under this path.
This directory contains two subdirectories: Production and Test for production c

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


src
main
java
ch.fhnw.ima.intro
Calculator
SetupChecker
test
java
Reformat Code ⌥⌘L
Optimize Imports ^⌥O
Delete... ⌦
Make Module 'prog-II-exercise-01-intro'
Compile 'SetupChecker.java' ⇧⌘F9
Run 'SetupChecker.main()' ^⇧R
Debug 'SetupChecker.main()' ^⇧D

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.

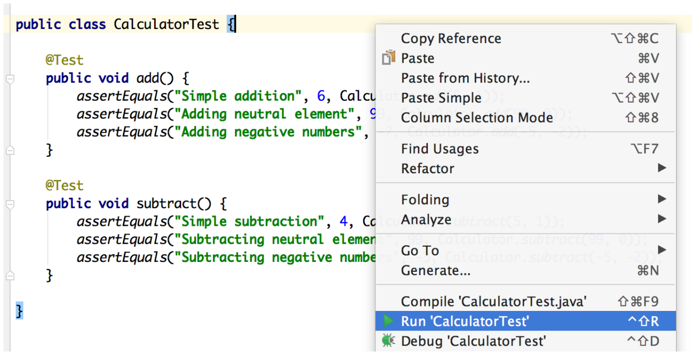
JUnit is a wide-spread framework to write unit tests in Java. Generally speaking, unit tests assert that a unit of code behaves according to expectations (in Java, a unit corresponds to a class). Unit testing has a lot of benefits and learning to use JUnit is well worth the effort.

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:



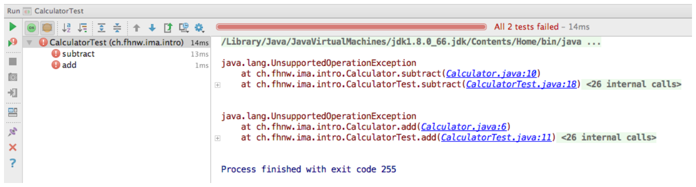
```
public class CalculatorTest {

    @Test
    public void add() {
        assertEquals("Simple addition", 6, Calcu
        assertEquals("Adding neutral element", 9
        assertEquals("Adding negative numbers",
    }

    @Test
    public void subtract() {
        assertEquals("Simple subtraction", 4, Ca
        assertEquals("Subtracting neutral elemen
        assertEquals("Subtracting negative numbe
    }

}
```

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



Run CalculatorTest
All 2 tests failed – 14ms
CalculatorTest (ch.fhnw.ima.intro) 14ms
subtract 13ms
add 1ms

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_66.jdk/Contents/Home/bin/java ...

java.lang.UnsupportedOperationException
	at ch.fhnw.ima.intro.Calculator.subtract(Calculator.java:10)
	at ch.fhnw.ima.intro.CalculatorTest.subtract(CalculatorTest.java:18) <26 internal calls>


java.lang.UnsupportedOperationException
	at ch.fhnw.ima.intro.Calculator.add(Calculator.java:6)
	at ch.fhnw.ima.intro.CalculatorTest.add(CalculatorTest.java:11) <26 internal calls>


Process finished with exit code 255
```

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:

```java
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:

```java
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:

```java
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:

```java
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:

```java
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:

```java
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 (`-`):

```java
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

Chapter 4. Object-Oriented Programming

The chapter on Data Structures explained how data can be stored in collections. A `List<String>` e.g. allows to keep track of text values. But what if we want to store more heterogeneous values? We cannot use a `List` to keep track of a person’s name, a phone number, and a city or residence. Even if we could stuff a mixture of text values and numbers into a list, we would actually prefer a `List<Person>`, right? But unfortunately, the Java standard library does not offer a `Person` class.

The following sections explain how we can write classes like `Person` ourselves. Classes represent the core building blocks in Java. As an object-oriented language, Java uses the concepts of *composition* and *inheritance* to create programs out of these building blocks.

4.1. Classes & Objects

In the real world, we tend to group *things* into different categories, e.g. "Emma" is a person, while "Milo" is a dog, and "Basel" is a city. In Java, person, dog, and city are known as *classes*, while "Emma", "Milo", and "Basel" are *objects*.

Taking the city as an example, suppose you would like to build a catalogue of cities. Each city can be described by a name and a ZIP code. The following code shows how to model this data structure with a custom `City` class:

```java
public final class City { ①

    private final String name; ②
    private final int zipCode; ②

    public City(String name, int zipCode) { ③
        this.name = name;
        this.zipCode = zipCode;
    }

    public String getName() { ④
        return name;
    }

    public int getZipCode() { ④
        return zipCode;
    }

}
```

① The class declaration

② Two *fields* (sometimes also known as *instance variables* or *properties*)

③ A *constructor* (accepting two *parameters* or *arguments*, assigning them to the *fields*)

④ Two *methods* (specifically, two *accessor* methods)

The `public`/`private` and `final` keywords ensure that the class is well encapsulated and immutable. These aspects are not important in small programs. In larger applications, they prevent certain categories of errors and make the code base more maintainable.

Let's see how the `City` class can be used – or in Java lingo: how it can be instantiated.

The terms *"creating an object of"*, *"instantiating"*, or *"creating an instance of"* are all synonymous.

```java
City basel = new City("Basel", 4000);
City zurich = new City("Zurich", 8000);
City muttenz = new City("Muttenz", 4132);

System.out.println(basel.getName());      // prints "Basel"
System.out.println(muttenz.getZipCode()); // prints 4132
```

The `new` keyword creates an instance of a class by calling the *constructor* which matches the number and types of the given *arguments*.

4.2. Composition

Our `City` class is composed of two fields, a `name` and a `zipCode`. The `name` field is of type `String`, which is itself a class. This is the simplest form of class re-use: We built a new class `City` by composing it of an existing building block, the `String` class.

The same approach can be used with custom classes. The following class defines a `Person` by re-using our own `City` class:

```
public class Person {

    private final String firstName;
    private final String lastName;
    private final City city;

    public Person(String firstName, String lastName, City city) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.city = city;
    }

    public String getFirstName() {
        return firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public City getCity() {
        return city;
    }

}
```

When creating a `Person` instance, an existing `City` instance is passed to the `Person` constructor:

```
City basel = new City("Basel", 4000);
Person gigi = new Person("Gisela", "Oeri", basel);

System.out.println(gigi.getFirstName());          // prints "Gisela"
System.out.println(gigi.getCity().getZipCode()); // prints 4000
```

The `getCity().getZipCode()` chain reflects the hierarchical nature of class composition.

Now our building blocks are complete enough to address the initial problem, the wish for a list of persons:

```
City basel = new City("Basel", 4000);
Person gigi = new Person("Gisela", "Oeri", basel);
Person eva = new Person("Eva", "Herzog", basel);

List<Person> persons = Arrays.asList(gigi, eva);
```

4.3. Inheritance

As seen in the previous section, the technique of **composition** is used to model *"has a"* relationships: A person *has a* city of residence, a city *has a* name. This section will explain **inheritance**, a technique to express an *"is a"* relationship.

Starting from our `Person` class, we can declare a new class `Student` which *is a* special kind of person:

```
public class Student extends Person { ①

    private final int semester; ②

    public Student(String firstName, String lastName, City city, int semester) { ③
        super(firstName, lastName, city);  ④
        this.semester = semester; ⑤
    }

    public int getSemester() { ⑥
        return semester;
    }

}
```

① The `extends` keyword specifies that `Student` inherits all aspects of a `Person`

② A *field* (in addition to all fields inherited from `Person`)

③ A *constructor* to create `Student` instances

④ A call to the *super constructor* (which initializes the `Person` base class)

⑤ Initialization of the `semester` field from the constructor argument

⑥ An *accessor method* to the `semester` field

In this inheritance hierarchy, `Person` is referred to as the *base-class*. `Person` is the *super-class* of `Student`, while `Student` is a *sub-class* of `Person`.

Using the `Student` class is straight forward. Instances of `Student` offer all functionality of the base-class `Person`, **plus** additional functionality like the `getSemester()` accessor:

```java
City city = new City("Wherever", 9999);
Student harry = new Student("Harry", "Potter", city, 3);

System.out.println(harry.getFirstName()); // prints "Harry"
System.out.println(harry.getSemester());  // prints 3
```

One thing to note is that the class inheritance hierarchy defines how objects can be assigned to differently typed variables. The *"is a"* metaphor is a good indication of whether an assignment is allowed:

```java
City city = new City("Wherever", 9999);
Student harry = new Student("Harry", "Potter", city, 3);
Person bill = new Person("Bill", "Gates", city);

Person person;
Student student;

person = bill;       // bill is a person
person = harry;      // harry is a person

student = harry;     // harry is a student
// student = bill;   // DOES NOT COMPILE - bill is NOT a student
```

While it is very convenient to inherit the properties and behavior of a base-class, it is sometimes desirable to deviate in certain aspects. The concept of **overriding** allows a sub-class to change the behavior of certain methods. Let's change the `Student` class to return last names with a " (s)" suffix:

```java
public class Student extends Person {

    private final int semester;

    public Student(String firstName, String lastName, City city, int semester) {
        super(firstName, lastName, city);
        this.semester = semester;
    }

    public int getSemester() {
        return semester;
    }

    @Override
    public String getLastName() {
        return super.getLastName() + " (s)";
    }

}
```

The `@Override` marker is called an *annotation*. It marks the intention of overriding a method in the base-class. If the `Person` class was changed to no longer offer a `getLastName()` method, this annotation would allow the compiler to detect this inconsistency.

4.3.1. The Object root class

Let's look at a simple class which does not explicitly extend another class:

```java
public final class Dog {

    private final String name;

    public Dog(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

}
```

In Java, all classes implicitly extend the `Object` class. In other words, even though `Dog` does not declare an inheritance relationship via `extends`, it still automatically `extends Object`.

> Class `Object` is the root of the class hierarchy. Every class has `Object` as a superclass. All objects, including arrays, implement the methods of this class.
>
> — Java API documentation

Thus, all of your classes automatically inherit 11 methods declared by the `Object` class. Most of these methods are irrelevant in basic Java programs, but the `toString()` method is important, so let's look at it in further detail.

The `toString()` method returns a textual representation of an object. You can call `toString()` like any other method, but most notably, it is automatically invoked whenever you pass an object to `System.out.println()`:

```java
Dog dog = new Dog("Milo");
System.out.println(dog); // prints 'ch.fhnw.ima.oop.Dog@511d50c0'
```

As illustrated in the above example, the base implementation returns weird looking gibberish (it is a mixture between the name of the class and its hash code, but that's not important for the time being). Since `toString()` is a normal method, you are free to override it and provide a better textual representation of your object:

```java
public final class Dog {

    private final String name;

    public Dog(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    @Override
    public String toString() {
        return "Dog (Name: " + name + ")";
    }

}
```

The generated output now looks much friendlier:

```
Dog dog = new Dog("Milo");
System.out.println(dog); // prints 'Dog (Name: Milo)'
```

4.4. Enum Types

Now that you know how to define your own classes via composition and/or inheritance, let's have a look at a special kind of data modeling problem for which Java provides a simple solution.

Imagine a game like Pac-Man, where a character is moving either up, right, down, or left. Somewhere in your game, you are tracking the character's direction. What class or data-type would you choose for that purpose?

You might be tempted to use an `int` type and declare 4 named constants:

```
// BAD SOLUTION

public static final int UP = 0;
public static final int RIGHT = 1;
public static final int DOWN = 2;
public static final int LEFT = 3;

private int direction;

public void setDirection(int direction) {
    this.direction = direction;
}
```

This is quite readable, but very error prone: The type system can't prevent usage of a "wrong" value. Consequently, a program that compiles just fine would break at runtime:

```
setDirection(UP); // ok
setDirection(42); // oops
```

What we would need is a type that just allows the four values `UP`, `RIGHT`, `DOWN`, `LEFT` and nothing else. Java provides a special-purpose type to represent such fixed enumerations, the `enum` type:

```
// GOOD SOLUTION

public enum Direction { ①

    UP, RIGHT, DOWN, LEFT ②

}
```

① The enum keyword makes `Direction` extend the `Enum` base class

② All values are constants and are thus, by convention, in uppercase letters

Usage of an enum is straight-forward:

```
private Direction direction;

public void setDirection(Direction direction) {
    this.direction = direction;
}
```

Note how enums improve code readability: Just by looking at the type of the `direction` variable we already know something about its meaning (which was not the case with `int` constants). Java is a typed language. By using proper types to model our data we are taking advantage of the compiler. Errors can be caught at compile time and don't cause bugs at runtime. By using the `Direction` enum, it is impossible to create a compiling program that contains a "wrong" direction value:

```
setDirection(Direction.UP);
setDirection(Direction.DOWN);
// setDirection(42); IMPOSSIBLE (does not compile)
```

Each `enum` type automatically offers a variety of methods. Most importantly, the `values()` method returns all possible constants, and the `name()` method returns a constant's name:

```
for (Direction direction : Direction.values()) {
    System.out.println(direction.name());
}
```

If you're getting bored or overwhelmed by all this reading, you might want to write some code for a change. At this point you know just enough about object-oriented programming to get started with the first exercise.

4.5. Interfaces

Suppose you were writing a fun little game in which unicorns played a major role. You picked up the new concepts explained in the previous sections and came up with this nice and simple class:

```java
public class Unicorn extends Horse {

    private final String name;

    public Unicorn(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void glowInRainbowColors() {
        System.out.println("Glitter");
    }

}
```

As you can see, someone else already wrote a `Horse` class – you were able to inherit most of its functionality, keeping your new `Unicorn` class nice an simple. Since you were done so quickly, you put all your energy and focus into listing your lovingly named unicorns in a fancy way:

```
~ •*¨*• ~ ~ •*¨*• ~
       Mystery
       Rainbow
       Jessica
~ •*¨*• ~ ~ •*¨*• ~
```

This is the code that you wrote to generate the output:

```java
public static void main(String[] args) {
    List<Unicorn> unicorns = Arrays.asList(
            new Unicorn("Mystery"),
            new Unicorn("Rainbow"),
            new Unicorn("Jessica"));
    System.out.println(fancyUnicornNames(unicorns));
}

private static String fancyUnicornNames(List<Unicorn> unicorns) {
    StringBuilder fancyNames = new StringBuilder();
    fancyNames.append("~ •*¨*• ~ ~ •*¨*• ~\n");
    for (Unicorn unicorn : unicorns) {
        fancyNames.append("       ").append(unicorn.getName()).append("       \n");
    }
    fancyNames.append("~ •*¨*• ~ ~ •*¨*• ~");
    return String.valueOf(fancyNames);
}
```

Unfortunately, life is not all rainbows and unicorns. Soon you were forced to extend your game and introduce dragons. Dragons are not related to horses nor unicorns, thus you couldn't extend any existing class. For a second you were tempted to introduce a common base-class (e.g. LivingCreature) in which you could nicely handle the name field and the getName() accessor in order not to duplicate this code. However, Unicorn already extends Horse, and multiple inheritance is not supported in Java. You decided to bite the bullet and write a Dragon class from scratch:

```java
public class Dragon {

    private final String name;

    public Dragon(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void fly() {
        System.out.println("Flying higher and higher");
    }

    public void spitFire() {
        System.out.println("Sparks & Flames!");
    }

}
```

But wouldn’t it be nice if the list of dragon instances could be output in the same fancy style as the unicorns? Some quick copy/pasting, a bit of renaming – tada!

```java
public static void main(String[] args) {
    List<Dragon> dragons = Arrays.asList(
            new Dragon("Toothless"),
            new Dragon("Stormfly"),
            new Dragon("Hookfang")
    );
    System.out.println(fancyDragonNames(dragons));
}

private static String fancyDragonNames(List<Dragon> dragons) {
    StringBuilder fancyNames = new StringBuilder();
    fancyNames.append("~ •*¨*• ~ ~ •*¨*• ~\n");
    for (Dragon dragon : dragons) {
        fancyNames.append("       ").append(dragon.getName()).append("       \n");
    }
    fancyNames.append("~ •*¨*• ~ ~ •*¨*• ~");
    return String.valueOf(fancyNames);
}
```

Hopefully you noticed how `fancyUnicornNames` and `fancyDragonNames` are almost identical. While copy/pasting might still work for two methods, it obviously doesn't scale – soon enough you will end up with `fancyKnightNames`, `fancyFairyNames`, and `fancyMonsterNames`. The tiniest bug requires the same fix in 10 different places.

Taking a step back, things are actually really simple: Inside the `fancyNames` rendering code, you don't really care whether you output a unicorn's name, a dragon's name, or a monster's name. In other words, you don't really care whether your `fancyNames` method gets a `List<Unicorn>`, a `List<Dragon>`, or even a `List<Monster>`. All you care about, is that whatever is in the list has a `getName()` method.

4.5.1. Using an Interface to Create Reusable Code

Interfaces solve exactly this problem: They let you express a contract in order to write reusable code relying on that contract. In our case, the contract solely requires a `getName()` method:

```
public interface Named { ①

    String getName(); ②

}
```

① The `interface` keyword is used to define an interface

② A method signature (note the absence of a body, i.e. the method does not have a default implementation)

Now that the `Named` interface exists, we change the `Unicorn` and `Dragon` classes to implement it. Implementing an interface is like making an official promise to fulfill a contract. The compiler makes sure that this promise is actually kept. This is how the `Unicorn` class must be changed (same for `Dragon`):

```java
public class Unicorn extends Horse implements Named { ①

    private final String name;

    public Unicorn(String name) {
        this.name = name;
    }

    @Override  ②
    public String getName() {
        return name;
    }

    public void glowInRainbowColors() {
        System.out.println("Glitter");
    }

}
```

While extending multiple classes is not supported in Java, it would be perfectly legal to implement multiple interfaces.

Now the `fancyNames` method can be rewritten based on the new `Named` interface:

```
public class FancyNames {

    public static void main(String[] args) {
        List<Unicorn> unicorns = Arrays.asList(
                new Unicorn("Mystery"),
                new Unicorn("Rainbow"),
                new Unicorn("Jessica"));

        System.out.println(fancyNames(unicorns));

        List<Dragon> dragons = Arrays.asList(
                new Dragon("Toothless"),
                new Dragon("Stormfly"),
                new Dragon("Hookfang")
        );

        System.out.println(fancyNames(dragons));
    }

    private static <T extends Named> String fancyNames(List<T> namedItems) { ①
        StringBuilder fancyNames = new StringBuilder();
        fancyNames.append("~ •*¨*• ~ ~ •*¨*• ~\n");
        for (Named named : namedItems) { ②
            fancyNames.append("       ").append(named.getName()).append("       \n"); ③
        }
        fancyNames.append("~ •*¨*• ~ ~ •*¨*• ~");
        return String.valueOf(fancyNames);
    }

}
```

① Accepts a list with items that implement the `Named` interface [1: Note that simply accepting a `List<Named>` would not work. We are getting into complicated type parameterization territory here, but the quick explanation goes like this: The `Named` interface creates a relationship between the `Unicorn` and the `Dragon` class. It would seem natural that this also implies a relationship between a `List<Unicorn>` and a `List<Dragon>`. Unfortunately, these two types are not related from the Java compiler's point of view. Therefore, the item type `T` must explicitly be declared as `T extends Named`. Alternatively, an upper bounded wildcard could be used (`List<? extends Named>`). Further details are e.g. explained in the official Wildcards and Subtyping tutorial.]

② Iteration logic relies on the fact that all list items implement the `Named` interface

③ Actual usage of `getName()`, as declared by the `Named` interface

To summarize: Introducing the `Named` interface helped implementing the `fancyNames` method in a way that is reusable for unicorns, dragons and all future implementations of the `Named` interface.

4.5.2. Using an Interface to Achieve Encapsulation

Remember the `List`, `Set`, and `Map` interfaces from the introductory section on Data Structures? All of them are perfect illustrations of another important aspect of interfaces: Interfaces allow to hide implementation details. In simple words, they separate **what** can be done (e.g. the `List` interface) from **how** it is done (e.g. the `ArrayList` implementation). This separation makes programs easier to understand and maintain. The complexity of the **how** is hidden behind the interface, making it e.g. possible to program against the `List` interface without ever having to deal with the array-related complexity encapsulated in the `ArrayList` class. In the Data Structures section you were encouraged to always code to the `List` interface rather than the `ArrayList` implementation. By now it should be clear that this has at least two benefits:

- Oracle is completely free to re-implement, optimize, or otherwise change the `ArrayList` class – as long as it still properly implements the `List` interface, no changes will be required in your code.
- On the other hand, **you** are also completely free to switch to another `List` implementation – if you want to use a `LinkedList` rather than an `ArrayList`, you only need to swap out the implementation in one spot.

4.5.3. Working with Interfaces in Java 8

Everything you learned about interfaces so far is deeply baked into the Java language and works with versions as old as Java 5. With the latest version, Java 8, some very attractive language changes were made. At least two of them make working with interfaces even more pleasurable:

Default Methods

Traditionally, interfaces do not contain method bodies, but only declare method signatures:

```
public interface Named { ①

    String getName(); ②

}
```

① The `interface` definition

② A method signature without body

With Java 8 it is possible to provide default implementations for interface methods:

```
public interface Named {

    default String getName() {
        return "Unknown";
    }

}
```

A class implementing this interface then becomes very minimal:

```
public class Creature implements Named {

}
```

It fulfills the interface contract because it automatically inherits the default implementation, i.e. it is possible to call `getName` on a `Creature` instance:

```
Creature creature = new Creature();
System.out.println(creature.getName()); //  prints "Unknown"
```

Of course `Creature` could still override `getName` as if no default implementation existed.

Lambda Expressions

Let's revisit the `Named` interface without the default method implementation again:

```
public interface Named { ①

    String getName(); ②

}
```

① The `interface` definition

② A method signature without body

Methods without a body are called **abstract** methods, and interfaces with a single abstract method are called **SAM types** (Single Abstract Method types).

Lambda expressions, a new feature in Java 8, represent a convenience construct to make working with SAM types simpler:

```
Named myNamed = () -> "Lara Croft";
```

Note that the `Named myNamed` part is a standard variable type/name declaration, only the part to the right of the `=` sign is the lambda expression. Let's look at another example to get a better feel for SAM types and lambdas:

```
public interface Calculation {

    double calc(double a, double b);

}
```

`Calculation` again has a single, abstract method. Consequently, a lambda expression can be used to provide an implementation:

```
Calculation addition = (a, b) -> a + b;
```

A lambda expression has everything that a method has: an argument list, which is the `(a, b)` part, and a body, which is the `a + b` part after the arrow symbol. Just like methods, lambda expressions allow to encapsulate not just simple one-liners, but also more complex constructs:

```
Calculation fancy = (x, y) -> {
    double fx = Math.sin(x);
    double fy = Math.cos(y);
    return fx + fy;
};
```

Curly braces and a `return` statement are needed for lambda expressions that encompass more than one statement.

Up to now we have only looked at how lambda expressions can be defined. We will now look at how they are used.

Actually, you have already met lambda expressions in an earlier section, namely when working with the The Java 8 Stream API:

```
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]
```

The `map` method takes the `java.util.function.Function` SAM type as its argument, for which we provide an ad-hoc implementation in the form of the `word → word.length()` lambda expression.

Lambda expressions allow to pass code fragments around like data, typically as arguments to other methods.

4.6. Resources

Java4Kids: Chapter 3

https://yfain.github.io/Java4Kids/#_meet_classes_the_main_language_constructs

Java4Kids: Chapter 5

https://yfain.github.io/Java4Kids/#_interfaces_lambdas_abstract_and_anonymous_classes

Angelika Langer: Lambda Tutorial

http://www.angelikalanger.com/Lambdas/Lambdas.pdf

Chapter 5. Functional Programming

For most humans, new concepts are easier to understand if they can be related to things they already know. One of the appeals of object-oriented programming (OOP) is that objects have obvious analogies in real life. Persons, dogs, and cities are very familiar concepts, and it's easy to grasp how a `Person` class models certain aspects of a real-life person.

Unfortunately, concepts like assignments or method calls often have no analogy in real life. Functional programming (FP) is a programming style that addresses this problem: Just like OOP focuses on the concept of objects, functional programming (FP) focuses on the concept of mathematical functions to describe computer programs. Because most of us know math better than we know programming, FP concepts are often easy to understand because they are so similar to math.

Note that OOP and FP are not exclusive: Certain aspects of a program are best expressed with objects, while others are easier to program using functions. Java 8 is a perfect language to combine the best of two worlds! Let's see what FP brings to the table…

5.1. Mathematical Functions

If you look at this simple Java assignment through the eyes of a mathematician, it is utter nonsense:

```
x = x + 1
```

Similarly, the following Java method doesn't have anything to do with math:

```
int add(int a) {
  int c = a + this.b;
  this.b = c;
  return c;
}
```

To put them in contrast, let's look at two typical math functions and explore their common characteristics:

```
f(x) = sin(x)
```

```
f(a,b) = a + b
```

These functions

1. have a set of input (`x`, `a` and `b`)

2. each have one output
3. given the same input, always produce the same output

All three points sound really obvious in the context of mathematical functions. Yet, our Java methods often diverge from these rules in many ways.

Let’s look at our initial example again:

```
int add(int a) {
  int c = a + this.b;
  this.b = c;
  return c;
}
```

Ideally, a Java method declares its input and output as part of its signature. Looking at

```
int add(int a)
```

could give the impression that our example method has one input `a` and one output (the return value). But this is not the case! In fact, this method

1. pretends to have one input (`a`) but actually has two (`a` and `this.b`)
2. pretends to have one output (the return value) but actually has two (assigning `this.b = c`)
3. given the same input `a`, may produce a different result (depending on the value of `this.b`)

Obviously, it is not similar to a math function at all, which makes it really hard to understand.

The method has **side effects**: It reads values which are not declared as input parameters and writes values in addition to returning a result. Its these side effects that make the method hard to understand and cause it to deviate from the math function characteristics.

The following `add` method is much easier to read because it has no side effects and is thus similar to a math function:

```
static int add(int a, int b) {
  return a + b;
}
```

The `static` keyword makes it explicit that the function does not read nor write any fields

To summarize: In functional programming, methods explicitly declare their input parameters, return one result, and do not rely on side effects.

5.2. Programs as Data Pipelines

Let’s look at another example method:

```
public static int countFourLetterWords(File file) {
    List<String> words = readWordsFromFile(file);
    List<String> fourLetterWords = filterByLength(words, 4);
    int count = count(fourLetterWords);
    return count;
}
```

You probably have a pretty good understanding of what’s going on here, right? Even though the methods `readWordsFromFile`, `filterByLength`, and `count` are not shown in detail, it is still quite obvious what they are doing. This is due to the fact that all of them have good names, very explicit input parameters, return one result, and don’t rely on side effects.

Functional programming uses functions as the fundamental building blocks of a program. Just like in mathematics, where simple functions can be combined to express more complex calculations, Java methods can be combined to form a pipeline. Data flows through the method calls until the final result is returned.

Functional programming obviously makes code very readable.

5.3. Preventing Programming Errors

Let’s go back to our initial `add` example:

```
int add(int a) {
  int c = a + this.b;
  this.b = c;
  return c;
}
```

This example may seem a little far fetched and contrived – seriously, who writes cryptic code like this?!

Actually, side effects occur very often in object-oriented programming! Accessor and mutator methods like `getName()` and `setName(String name)` **only** work because of side effects. Fortunately, setters and getters are so easy to implement that there’s no need to get rid of their side effects completely (though some FP purists disagree on this).

In general, side effects in non-trivial methods make implementations hard to read but also hard to write correctly. The following example will illustrate how removing side effects can prevent programming errors.

Let’s dive in with a `Container` class, which models a vessel with contents and a maximal capacity:

```java
public class Container {

    private static final int CAPACITY = 100;

    private int contents;

    public Container(int contents) {
        this.contents = contents;
    }

    public int getContents() {
        return contents;
    }

    public void add(int amount) {
        if (contents + amount > CAPACITY) {
            throw new IllegalArgumentException("Exceeding capacity");
        }
        contents += amount;
    }

    public void remove(int amount) {
        if (contents < amount) {
            throw new IllegalArgumentException("Insufficient contents");
        }
        contents -= amount;
    }

    public void transferTo(int amount, Container toContainer) {
        remove(amount);
        toContainer.add(amount);
    }

}
```

Note how `add` and `remove` both alter the container’s contents as a side effect. Moreover, both may throw an exception, another form of side effect. Specifically, `add` checks that the container cannot exceed its capacity, while `remove` assures that the requested amount is limited by the current contents.

The method `transferTo` uses `add` and `remove` to transfer a specific `amount` from this container to another container.

This is how the `Container` class can be used:

```
Container a = new Container(80);
Container b = new Container(80);
// transfer 20 from container a to container b
a.transferTo(20, b);
System.out.println(a.getContents());    // prints 60
System.out.println(b.getContents());    // prints 100
```

Interesting things happen when an impossible transfer is requested:

```
Container c = new Container(80);
Container d = new Container(80);
try {
    // the following call will fail
    c.transferTo(70, d); ①
} catch (IllegalArgumentException e) {
    System.out.println(e.getMessage()); // prints "Exceeding capacity" ②
}
System.out.println(c.getContents());    // prints 10 ③
System.out.println(d.getContents());    // prints 80
```

① An impossible transfer which exceeds capacity of target container `d`

② As expected, an exception is thrown by the nested call to `add`

③ **Programming Error**: Even though the transfer failed, the amount is still deduced from container `c`

Of course we could invest time in fixing the `transferTo` implementation and making sure not to alter container states in case of errors. But it is a matter of fact that the various side effects make this implementation non-trivial. Instead of wasting time and energy, let's instead look at how the same problem could be solved in a much more robust way – functional programming to the rescue!

For starters, we will change the `add` and `remove` methods to no longer alter the contents by side effect, but instead return a new `Container` instance with the changed contents:

```java
public class Container {

    private static final int CAPACITY = 100;

    private final int contents;

    public Container(int contents) {
        this.contents = contents;
    }

    public int getContents() {
        return contents;
    }

    public Container add(int amount) {
        if (contents + amount > CAPACITY) {
            throw new IllegalArgumentException("Exceeding capacity");
        }
        return new Container(contents + amount);
    }

    public Container remove(int amount) {
        if (contents < amount) {
            throw new IllegalArgumentException("Insufficient contents");
        }
        return new Container(contents - amount);
    }

}
```

As a next step, we want to improve the transfer scenario. Specifically, we want to make its input and output much more explicit:

- A transfer needs two containers, a **from** container and a **to** container
- A successful transfer also returns two containers, the **from** container now has less contents, while the **to** container now has more contents

In other words, we need a pair of containers as input, and also return a pair of containers. Let's introduce a `Transfer` class which models a pair of containers:

```
public class Transfer {

    private final Container from;
    private final Container to;

    public Transfer(Container from, Container to) {
        this.from = from;
        this.to = to;
    }

    public Container getFrom() {
        return from;
    }

    public Container getTo() {
        return to;
    }

}
```

The new `transfer` method now looks like this:

```
public Transfer transfer(int amount, Transfer transfer) {
    Container updatedFrom = transfer.getFrom().remove(amount);
    Container updatedTo = transfer.getTo().add(amount);
    return new Transfer(updatedFrom, updatedTo);
}
```

Note that it only reads state from its input parameter. Consequently, it is decoupled from the `Container` object, which we can make explicit by moving it to its own class:

```
public class ContainerService {

    public static Transfer transfer(int amount, Transfer transfer) {
        Container updatedFrom = transfer.getFrom().remove(amount);
        Container updatedTo = transfer.getTo().add(amount);
        return new Transfer(updatedFrom, updatedTo);
    }

}
```

Again, using the `static` modifier makes it explicit that no instance state is read nor written.

This is how transfers can be made with our new design:

```
Container a = new Container(80);
Container b = new Container(80);
Transfer input = new Transfer(a, b);
Transfer result = ContainerService.transfer(20, input);
System.out.println(result.getFrom().getContents()); // prints 60
System.out.println(result.getTo().getContents());   // prints 100
```

How will it behave if an invalid transfer is requested?

```
Container c = new Container(80);
Container d = new Container(80);
try {
    Transfer input = new Transfer(c, d);
    // the following call will fail
    Transfer result = ContainerService.transfer(70, input); ①
} catch (IllegalArgumentException e) {
    System.out.println(e.getMessage()); // prints "Exceeding capacity" ②
}
System.out.println(c.getContents());    // prints 80 ③
System.out.println(d.getContents());    // prints 80
```

① An impossible transfer which exceeds capacity of target container `d`.

② Code flows to the `catch` clause, `result` will thus not be available (which makes sense).

③ **Fixed**: If the transfer fails, the amount is **no longer** deduced from container `c`

Phew, this was hard – but the code now works as expected! The functional programming principles of explicitly declaring input parameters and preventing side effects made the code more robust.

5.4. Map / Filter / Reduce

Often, functional programming is associated with the map/filter/reduce design pattern. We have met this pattern in The Java 8 Stream API, here it is again:

```
Stream<String> words = Stream.of("hello", "world", "java", "is", "cool");
long numberOfFourLetterWords = words.
        map(word -> word.length()).          // map
        filter(length -> length == 4).       // filter
        count();                             // reduce to single result
System.out.println(numberOfFourLetterWords); // prints 2
```

But what is so "functional" about streams? In fact, the same problem could also be solved without using streams:

```
List<String> words = Arrays.asList("hello", "world", "java", "is", "cool");
long numberOfFourLetterWords = 0;
for (String word : words) {
    if (word.length() == 4) {
        numberOfFourLetterWords++;
    }
}
System.out.println(numberOfFourLetterWords); // prints 2
```

This second style is known as imperative programming: Control structures like `for` and `if` determine how a sequence of statements is executed. In contrast, streams allow to express the same intentions **without** using control structures, solely by using functions. Extracting the lambda expressions to local variables makes this more visible:

```
Stream<String> words = Stream.of("hello", "world", "java", "is", "cool");

// A function from String -> Integer
Function<String, Integer> mapFunction = word -> word.length();
// A function from Integer -> Boolean
Predicate<Integer> filterFunction = length -> length == 4;

long numberOfFourLetterWords = words.
        map(mapFunction).
        filter(filterFunction).
        count();
System.out.println(numberOfFourLetterWords);
```

In summary, using the functional approach of map/filter/reduce allows programmers to focus on the heart of the computation rather than on the details of loops, branches, and control flow.

Because the details of loops, branches, and control flow are no longer controlled by the programmer, code written in a functional style is very suitable for parallel execution, e.g. on a distributed cluster (Google's MapReduce technology is the most prominent example).

5.5. Summary

Functional programming uses functions as the fundamental building blocks of a program. In Java, such functions are built by writing methods or lambda expressions which explicitly declare their input parameters and avoid any side effects. Functional programming makes programs easier to read and prevents certain types of errors.

5.6. Resources

Kris Jenkins: What Is Functional Programming?

http://blog.jenkster.com/2015/12/what-is-functional-programming.html

Google MapReduce

https://en.wikipedia.org/wiki/MapReduce